GUILHERME STUTZ TÖWS

PETRIGRAPH - UM ALGORITMO PARA PLANEJAMENTO POR DESDOBRAMENTO DE REDES DE PETRI

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Fabiano Silva

CURITIBA

2008

GUILHERME STUTZ TÖWS

PETRIGRAPH - UM ALGORITMO PARA PLANEJAMENTO POR DESDOBRAMENTO DE REDES DE PETRI

Dissertação aprovada como requisito parcial à obtenção do grau de Mestre no Programa de Pós-Graduação em Informática da Universidade Federal do Paraná, pela Comisso formada pelos professores:

Orientador: Prof. Dr. Fabiano Silva
Departamento de Informática, UFPR

Profa. Dra. Leliane Nunes de Barros
Instituto de Matemática e Estatística, USP

Prof. Dr. Marcos Alexandre Castilho
Departamento de Informática, UFPR

Curitiba, 21 de maio de 2008

Universidade Federal do Paraná
Setor de Ciências Exatas
UFPR Programa de Pós-graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Guilherme Stutz Töws, avaliamos o trabalho intitulado, "*PETRIGRAPH – UM ALGORITMO PARA PLANEJAMENTO POR DESDOBRAMENTO DE REDES DE PETRI*", cuja defesa foi realizada no dia 21 de maio de 2008, às 10:00 horas, no Auditório do Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 21 de maio de 2008.

Prof. Dr. Fabiano Silva
DINF/UFPR – Orientador

Profª. Dra. Leliane Nunes de Barros
IME/USP – Membro Externo

Prof. Dr. Marcos Alexandre Castilho
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

AGRADECIMENTOS

Gostaria de agradecer a todas as pessoas que tornaram esta conquista possível.

Orientador Prof. Dr. Fabiano Silva, pelo contínuo apoio e orientação desde a conceitualização da pesquisa até a sua realização.

Prof. Dr. Luis Allan Kunzle, que praticamente co-orientou o trabalho e providenciou apoio fundamental na formatação e correção deste texto.

Prof. Dr. Marcos Alexandre Castilho e Profa. Dra. Leliane Nunes de Barros, pelo extenso corpo de sugestões feitas para a melhoria do trabalho.

Todos os outros professores do curso de Mestrado, que esforçaram-se para compartilhar de seus conhecimentos de ciência.

Colegas do grupo de pesquisa do Laboratório de Inteligência Computacional, que compartilharam referências e material indispensável para a estruturação deste texto.

Meus pais, João Guilherme e Angela Maria, e minha irmã Florence, pelo apoio material e emocional, e modelo de comportamento e ética.

Familiares e amigos, pelo contínuo suporte e compreensão.

Deus, pela graciosa doação de recursos e material de pesquisa.

RESUMO

É proposto um algoritmo, chamado *Petrigraph*, que é uma extensão do algoritmo *Petrify* de Hickmott *et al* [1]. O algoritmo Petrify apresenta uma solução ao problema de planejamento baseado na conversão do problema em uma rede de Petri e análise de alcançabilidade sobre esta rede por meio da técnica de desdobramento.

O *Petrigraph* inova em utilizar informações da representação do problema na linguagem PDDL para otimizar o processo de criação da rede de Petri através da criação de variáveis multivaloradas, que é uma técnica utilizada para reduzir o espaço de estados de problemas derivados do PDDL, e grafos de transição de domínio, que proporcionam a conexão entre variáveis multivaloradas e redes de Petri seguras.

Após a revisão dos conceitos fundamentais de redes de Petri e planejamento, descreve-se o funcionamento do *Petrigraph*, como ele se compara com o *Petrify*, e como o algoritmo foi implementado.

São mostrados resultados de testes feitos com o algoritmo *Petrigraph*, comparando-o ao *Petrify* e ao *SatPlan* em sete domínios extraídos de competições de planejamento. É demonstrado que, sob certas condições, este método resulta num processo de desdobramento mais rápido e um planejador mais rápido que o algoritmo *Petrify* original.

ABSTRACT

This work improves the Petrify technique, which presented a solution to the planning problem based on the Petri net network analysis technique for reachability analysis. Information from the PDDL encoding is used in order to optimize the process of acquiring a Petri net from a planning problem. This approach leads to a reduction in the size of the generated net. The method employs the creation of multi-valued variables, which are a technique used to reduce unnecessary state space for PDDL-derived problems, and domain transition graphs, which provide the bridging between planning problem and Petri net. It is demonstrated that, under certain conditions, this method results in a faster unfolding process and a faster planner than the original Petrify algorithm.

SUMÁRIO

LISTA DE FIGURAS

CAPÍTULO 1

INTRODUÇÃO

O Planejamento em Inteligência Artificial envolve a geração de seqüências de ações, ou planos, que transformam uma situação inicial em uma configuração da situação final que satisfaça um conjunto de objetivos interdependentes. Planejamento é uma ferramenta poderosa para a produção de *sistemas baseados em conhecimento* - sistemas capazes de formular planos de ação a serem executados a partir de um conhecimento genérico sobre tomada de decisões e informação fornecida sobre problemas específicos. Alguns exemplos de problemas de planejamento incluem tarefas de transporte e logística, organização de processos industriais, navegação automatizada, planejamento didático em sistemas tutores informatizados, planejamento do processo de resolução de problemas, entre outros.

O problema de alcançabilidade em planejamento clássico é PSPACE-completo [2]; assim, é necessário o estudo cuidadoso de algoritmos capazes de resolver este problema em tempo hábil. Um campo promissor, e que será o foco deste estudo, é o que relaciona planejamento em inteligência artificial com redes de Petri. O problema de alcançabilidade em redes de Petri também é PSPACE-completo [3], e tem a vantagem de ter décadas de pesquisa já desenvolvidas sobre ele, enquanto a pesquisa em planejamento tomou forma na década de 90.

O presente estudo foi iniciado a partir da análise do método recentemente desenvolvido por Hickmott *et al*, chamado *Petrify* [1]. O *Petrify* resolve o problema de alcançabilidade em planejamento através da transformação do problema de planejamento em uma rede de Petri equivalente, e resolvendo o problema de alcançabilidade nesta rede de Petri pelo algoritmo denominado desdobramento (*unfolding*) [4].

Foi verificado que o processo de conversão de problema de planejamento para rede de Petri gera uma rede de tamanho desnecessariamente grande, o que influe diretamente na velocidade do passo mais custoso, o de desdobramento. Neste estudo, utilizamos

o método de criação de variáveis multivaloradas [5] e grafos de transição de domínio [6] para criar uma rede menor, que conseqüentemente diminui o tempo de execução do desdobramento. Foi dado a este método o nome de *Petrigraph*, refletindo a sua criação como desenvolvimento do *Petrify* e utilização de grafos de transição de domínio.

O método *Petrigraph* tem a intenção de resolver problemas de planejamento do mesmo modo que o *Petrify*, ou seja, pela transformação em redes de Petri que são então desdobradas. A vantagem do método proposto é que a estrutura do problema formulado em PDDL[1] [7] é usada para remover partes da rede que não são necessárias para que esta rede seja considerada segura e própria para o desdobramento. Assim, a rede de Petri menor permite um desdobramento mais rápido.

O estudo neste trabalho utiliza o planejamento chamado *clássico*, com as extensões de tipagem e pré-condições negativas do PDDL (seção 3.3). Sugestões para estender a aplicação deste método a domínios com extensões de PDDL são discutidas no capítulo 6.

1.1 Organização do trabalho

O capítulo 2 apresenta uma revisão bibliográfica dos trabalhos relacionados com redes de Petri. O capítulo 3 revisa a bibliografia relacionada a planejamento, bem como os algoritmos que serão utilizados no *Petrigraph*. O capítulo 4 detalha a metodologia do *Petrigraph* e sua implementação. O capítulo 5 apresenta os resultados obtidos com este estudo e mostra análises e comparações entre o método proposto e métodos já conhecidos. No capítulo 6 são apresentadas as considerações finais sobre o trabalho, bem como propostas para trabalhos futuros.

[1] *Planning Domain Definition Language.*

CAPÍTULO 2

REDES DE PETRI

Neste capítulo são apresentados os principais conceitos relativos às redes de Petri (RdP), tais como a representação gráfica e o formalismo matemático desta teoria. A seção 2.4 é dedicada ao problema de alcançabilidade em RdP, um dos focos deste trabalho. São descritas algumas técnicas para se encontrar uma solução para este problema, destacando a técnica de desdobramento, que tem sido aplicada com sucesso neste tipo de caso.

A construção de um desdobramento é apresentada na seção 2.5 juntamente com definições necessárias para a compreensão do tema. A ferramenta *MOLE*, descrita na seção 2.5.1, é utilizada para gerar o desdobramento de uma determinada classe de redes de Petri.

2.1 Definições

O conceito de redes de Petri foi introduzido por Carl Adam Petri em sua tese de doutorado em 1962 [8] como uma teoria para descrever relações entre condições e eventos.

Redes de Petri são ferramentas matemáticas para modelar, analisar e projetar sistemas a eventos discretos, possibilitando a representação de propriedades tais como paralelismo, concorrência, controle, conflitos e sincronização [9].

Uma RdP consiste em um grafo dirigido, ponderado e bipartido contendo dois tipos de nós, chamados lugares e transições. Os nós são conectados por segmentos orientados chamados arcos. Resumidamente, os elementos de uma RdP são:

- **Lugares:** Representam estados parciais possíveis do sistema. Neste trabalho, são representados graficamente por círculos;
- **Transições:** Representam eventos. Uma transição tem um certo número de lugares de entrada e de saída, respectivamente pré-condições e pós-condições do evento.

Neste trabalho, são representados graficamente por barras retangulares.

- **Marcas:** Representam o estado parcial de um sistema. A presença de uma marca em um lugar indica que a proposição representada por aquele lugar é verdadeira, ou ainda a presença de uma ou mais marcas indica a quantidade de recursos disponíveis. Representados graficamente aqui como círculos preenchidos dentro dos lugares.

- **Arcos:** Conectam tanto lugares a transições como transições a lugares. Determinam o fluxo das marcas na rede. Cada arco tem um peso, representado com um número sobre a seta, que determina a quantidade de marcas consumidas ou produzidas pelas transições quando da ocorrência de eventos a que elas correspondem. Caso esse número não exista, o peso do arco é assumido como 1. Neste trabalho, são representados graficamente por setas apontando na direção do fluxo.

Formalmente, uma Rede de Petri pode ser definida como:

$$N = (P, T, F, W, M)$$

onde:

$$\begin{aligned}
P = \{p_1, p_2, \cdots, p_i\} &\quad \text{é um conjunto finito de lugares;} \\
T = \{t_1, t_2, \cdots, t_j\} &\quad \text{é um conjunto finito de transições;} \\
F \subseteq (P \times T) \cup (T \times P) &\quad \text{é um conjunto de arcos (relação de fluxo);} \\
W : F \to \mathbb{N} &\quad \text{é uma função de peso;} \\
M : P \to \mathbb{N} &\quad \text{é o número de marcas;} \\
P \cap T = \emptyset \text{ e } P \cup T \neq \emptyset &\quad P \text{ e } T \text{ são conjuntos disjuntos.}
\end{aligned}$$

A figura 2.1 mostra um exemplo desse tipo de rede.

Uma outra classe de rede de Petri muito utilizada é a Condição/Evento, brevemente definida a seguir:

Definição 1 (Rede Condição/Evento)**.** *Uma rede é chamada Condição/Evento quando W e M obedecem a seguinte restrição:*

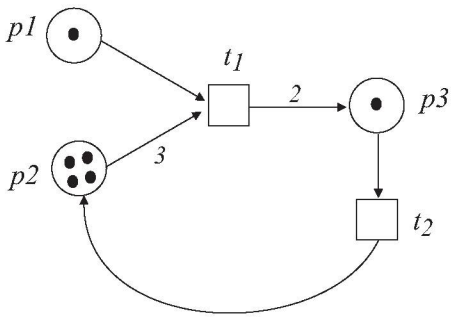


Figura 2.1: rede de Petri Lugar/Transição

$$M : P \rightarrow \{0, 1\}$$

$$W : F \rightarrow \{0, 1\}$$

Nesta rede as transições representam eventos, os lugares representam condições e uma marcação em um lugar indica que a condição está satisfeita.

A figura 2.2 ilustra uma rede Condição/Evento que simula o funcionamento de um semáforo. Existe uma marcação no lugar *vermelho*, o que indica que a condição *vermelho* está satisfeita na presente configuração.

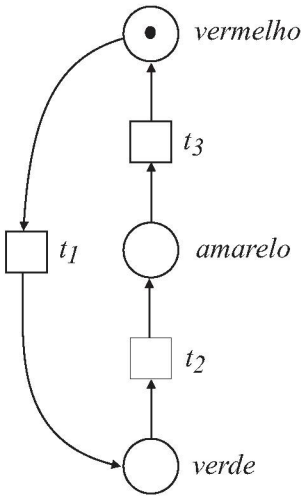


Figura 2.2: rede de Petri Condição/Evento

A relação de fluxo F é definida pelo pré-conjunto e pelo pós-conjunto dos lugares e

transições da rede. Para $x \in P \cup T$, o pré-conjunto de x, denominado $\bullet x$, e o pós-conjunto de x, denominado $x\bullet$, podem ser definidos como:

$$\bullet x = \{y \in P \cup T \mid W(\{y, x\}) \geq 1\}$$
$$x\bullet = \{y \in P \cup T \mid W(\{x, y\}) \geq 1\}$$

O comportamento de um sistema pode ser descrito pelas mudanças em seus estados. Um sistema modelado em uma rede de Petri simula essa dinâmica de acordo com uma regra de disparo de transição:

Definição 2 (Regra de disparo de transição)**.** *Uma transição t está habilitada, se seus lugares de entrada estiverem marcados com a quantidade de marcações maior ou igual ao valor do peso indicado no arco. Se a transição for disparada, ela remove a quantidade de marcas dos lugares de entrada e adiciona nos lugares de saída a quantidade de marcas igual ao valor do peso do arco entre a transição e os respectivos lugares de saída.*

Na rede da figura 2.1, a transição t_1 está habilitada, uma vez que os lugares de entrada, p_1 e p_2 estão marcados e satisfazem a regra. O resultado do disparo da transição t_1 pode ser visto na figura 2.3.

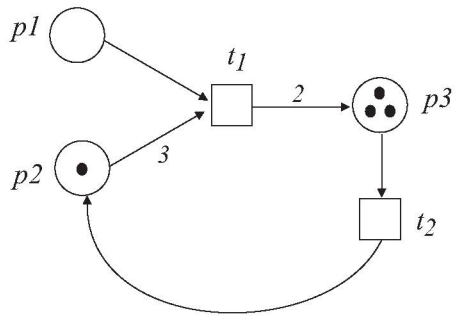


Figura 2.3: Rede da figura 2.1 após o disparo da transição t_1

O disparo da transição t_1 retira três das quatro marcas do lugar p_2, e uma marca do lugar p_1 e adiciona duas marcas no lugar de saída p_3.

2.2 Representação matricial

Uma rede de Petri também pode ser definida como um conjunto de matrizes de números naturais. O comportamento dinâmico destas matrizes é descrito por um sistema linear [10].

A relação de fluxo F e a função de peso W são agrupadas e armazenadas em matrizes que definem a ligação entre os lugares e as transições da rede.

A matriz de incidência de entrada `Pre` descreve os arcos orientados que ligam lugares a transições. $\texttt{Pre}(p,t)$ representa o peso do arco (p,t). Se $\texttt{Pre}(p,t) = 0$, então não existe um arco entre o lugar p e a transição t.

A matriz de incidência de saída `Pos` descreve os arcos orientados que ligam transições a lugares. $\texttt{Pos}(p,t)$ representa o peso do arco (t,p). Se $\texttt{Pos}(p,t) = 0$, então não existe um arco entre a transição t e o lugar p.

A partir das matrizes `Pre` e `Pos` calcula-se a matriz de incidência `C`, (`C` = `Pos` - `Pre`), que fornece o balanço das marcações na rede quando do disparo das transições.

As próximas definições serão exemplificadas utilizando a rede da figura 2.1. A representação matricial da rede traz as seguintes matrizes:

$$\texttt{Pre} = \begin{array}{c} \begin{matrix} t_1 & t_2 \end{matrix} \\ \begin{bmatrix} 1 & 0 \\ 3 & 0 \\ 0 & 1 \end{bmatrix} \end{array} \begin{matrix} p_1 \\ p_2 \\ p_3 \end{matrix} \qquad \texttt{Pos} = \begin{array}{c} \begin{matrix} t_1 & t_2 \end{matrix} \\ \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 2 & 0 \end{bmatrix} \end{array} \begin{matrix} p_1 \\ p_2 \\ p_3 \end{matrix} \qquad \texttt{C} = \begin{array}{c} \begin{matrix} t_1 & t_2 \end{matrix} \\ \begin{bmatrix} -1 & 0 \\ -3 & 1 \\ 2 & -1 \end{bmatrix} \end{array} \begin{matrix} p_1 \\ p_2 \\ p_3 \end{matrix}$$

Para representar a marcação da rede utiliza-se um vetor coluna M, cujo tamanho é a quantidade de lugares. $M(p)$ define o número de marcas em um lugar p, assim, $M(p_1) = 1$, $M(p_2) = 4$ e $M(p_3) = 1$. Portanto, a marcação inicial da rede é o vetor transposto $M_0 = [1\ 4\ 1]^T$.

Utilizando a notação matricial para a regra de disparo de transição, uma transição t está habilitada se, e somente se, $\forall p \in P,\ M(p) \geq \texttt{Pre}(p,t)$.

Se a transição for disparada, uma nova marcação será definida através da remoção de $\texttt{Pre}(p,t)$ marcações dos lugares de entrada e da adição de $\texttt{Pos}(p,t)$ marcações nos

lugares de saída. A nova marcação M' é obtida através de: $\forall p \in P,\ M'(p) = M(p) - \mathtt{Pre}(p,t) + \mathtt{Pos}(p,t)$.

Para efeito de simplificação, vamos utilizar a notação $\mathtt{Pre}(.,t)$ para a coluna da matriz $\mathtt{Pre}$ referente à transição t, e assim da mesma forma para as matrizes $\mathtt{Pos}$ e $\mathtt{C}$. Com isso, a equação para obter a nova marcação pode ser escrita como: $M' = M - \mathtt{Pre}(.,t) + \mathtt{Pos}(.,t)$ ou ainda $M' = M + \mathtt{C}(.,t)$.

Na rede exemplo, a marcação M' obtida após o disparo de t_1 sobre a marcação inicial M, ($M \xrightarrow{t_1} M'$) é encontrada a partir da equação: $M' = [1\ \ 4\ \ 1]^T + [\text{-}1\ \text{-}3\ \ 2]^T = [0\ \ 1\ \ 3]^T$ e pode ser vista na figura 2.3.

Para generalizar essa fórmula e calcular uma nova marcação após o disparo de uma seqüência de transições define-se o *vetor característico* $\bar{s}$ da seqüência s. Esse vetor é de tamanho igual ao número de transições da rede, sendo $\bar{s}(t)$ o número de vezes que a transição t foi disparada. A equação fundamental das redes de Petri é dada por:

$$M' = M + \mathtt{C}.\bar{s} \tag{2.1}$$

2.3 Propriedades de redes de Petri

As propriedades relativas às redes de Petri são divididas entre *estruturais*, que não dependem da marcação, e *comportamentais*, que estão relacionadas com as RdP marcadas. Neste trabalho, a atenção é voltada para a propriedade estrutural de aciclicidade e para as propriedades comportamentais: limitabilidade, reversibilidade e alcançabilidade.

Definição 3 (Rede de Petri acíclica). *Uma RdP é acíclica se nenhum subconjunto de seus arcos forma um ciclo direcionado, ou seja, se não contiver uma seqüência de lugares e transições* $\{x_1,x_2,\ldots,x_n,x_1\}$ *para* $n \geq 2$ *tal que todos os pares* (x_1,x_2), (x_2,x_3), ..., (x_{n-1},x_n), (x_n,x_1) *correspondam a arcos na rede.*

Definição 4 (Rede de Petri limitada). *Uma RdP é limitada, se para toda marcação, o número de marcas em cada lugar da rede é limitado. A rede é dita k-limitada quando a capacidade é limitada a k marcas. Se* $k = 1$, *a rede 1-limitada é chamada de rede segura*

(safe net).

$$\forall p \in P \;\; e \;\; \forall M \in A(N, M_0) \;,\; M(p) \leq k$$

Definição 5 (Rede de Petri reversível). *Uma RdP é dita reversível, ou reiniciável, se for possível a partir de qualquer marcação acessível encontrar uma seqüência de disparos s que leve a rede de volta à marcação inicial. Redes deste tipo são necessariamente cíclicas.*

$$\forall M \in A(N, M_0) \;,\; \exists s \mid M \xrightarrow{s} M_0$$

Definição 6 (Alcançabilidade). *O conjunto de marcações acessíveis $A(N, M_0)$ de uma RdP N com uma marcação inicial M_0 é o conjunto das marcações alcançáveis a partir da marcação inicial, através de uma seqüência de disparos s.*

$$A(N, M_0) = \{M_i | \exists s \; M_0 \xrightarrow{s} M_i\}$$

Quando o conjunto de marcações acessíveis é finito, pode ser representado sob a forma de um grafo, denominado *grafo de alcançabilidade* onde os nós são as marcações acessíveis e os arcos são as transições disparadas entre as marcações. A figura 2.4 ilustra todas as possíveis marcações e todos os possíveis disparos em cada marcação para a rede da figura 2.1; Por exemplo, a partir da posição inicial (marcada como 0 no grafo), pode-se passar à configuração 1 com uma marca em p_2 e três marcas em p_3, ou à configuração 5 com uma marca em p_1 e cinco marcas em p_2.

Saber se uma determinada marcação é alcançável a partir da marcação inicial da rede é um problema relevante na área. Essa verificação e a busca por uma seqüência de transições que leva da marcação inicial até a marcação desejada define o *problema de alcançabilidade em redes de Petri*, que será aprofundada na seção 2.4.

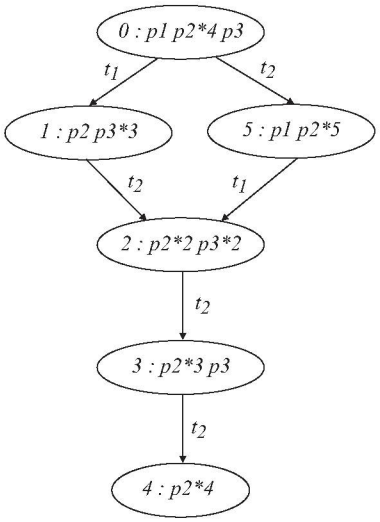


Figura 2.4: Grafo de alcançabilidade da rede da figura 2.1

2.4 O problema de alcançabilidade

A equação fundamental (2.1) pode ser utilizada para se determinar uma seqüência de transições s tal que $M_0 \xrightarrow{s} M$. Porém, a existência de um vetor que atenda a equação não é uma condição necessária para que a marcação M seja realmente alcançada a partir da marcação inicial, uma vez que a ordem dos disparos é perdida e a solução encontrada pode trazer vetores $\bar{s}$ que não correspondem a seqüências possíveis de disparos na rede.

O problema de alcançabilidade pode ser definido como o de verificar se uma dada marcação M é alcançável a partir da marcação M_0, ou seja, se:

$$M \in A(N, M_0)$$

É preciso destacar que muitas vezes essa exata marcação que está sendo buscada não será encontrada e sim uma marcação maior que irá contê-la. Chamamos M_s à marcação que está contida em uma marcação alcançável M ($M_s \subseteq M$), ou ainda, que $M_s \leq M$. Temos então o problema de alcançabilidade de sub-marcação, definido como o problema de verificar se existe um M tal que:

$$M \in A(N, M_0) \text{ onde } M_s \subseteq M$$

O problema de alcançabilidade de sub-marcação é teoricamente equivalente ao problema de alcançabilidade, que se sabe ser um problema decidível usando espaço exponencial [11]. Sua complexidade computacional está em aberto na citação mais recente encontrada para o caso geral [12], mas sabe-se que o problema de alcançabilidade em redes de Petri acíclicas é NP-Completo [13] e em redes k-limitadas é PSPACE-Completo [3].

Silva em seus trabalhos [14] e [15] utilizou técnicas de programação inteira sobre a equação fundamental das redes de Petri (2.1) para resolver problemas de alcançabilidade em redes acíclicas. Sua abordagem foi fundamentada no fato da equação fundamental ser uma condição necessária e suficiente para alcançabilidade em redes de Petri acíclicas [9].

O grafo de alcançabilidade, descrito na seção 2.3, é utilizado para resolver problemas de alcançabilidade envolvendo métodos de busca e técnicas heurísticas, mas pode ser usado apenas para redes pequenas devido à explosão do espaço de estados.

A técnica de desdobramento [4, 16] tem sido aplicada com sucesso para resolver problemas de alcançabilidade, reduzindo significativamente o tamanho do espaço de estados, comparado com o método do grafo de alcançabilidade.

2.5 Desdobramento

O conceito de desdobramento (*Unfolding*) foi introduzido originalmente por McMillan [16] com o objetivo de evitar o problema da explosão do espaço de estados na análise dos sistemas modelados em redes de Petri cíclicas.

A técnica é um método para resolver o problema de alcançabilidade que explora e preserva as informações de concorrência da rede. O desdobramento de uma rede cíclica é uma outra rede, acíclica, finita e que possui todas as marcações alcançáveis da rede original.

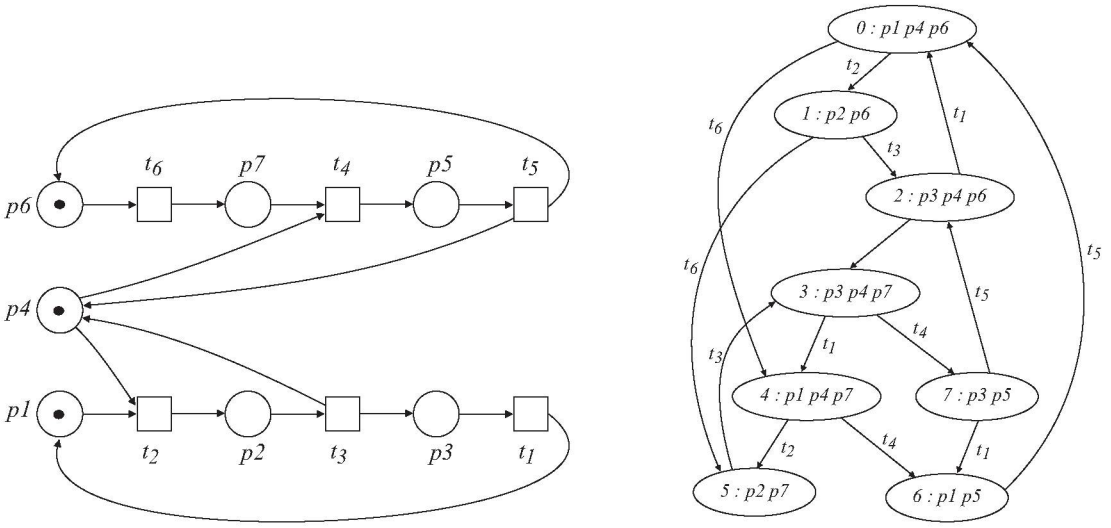
Para o entendimento da técnica de desdobramento é necessário definir os seguintes termos relativos a redes de Petri:

Definição 7 (Relação de ordem). *Seja* $N = (P, T, F, W, M)$ *uma rede acíclica e seja* $x_1, x_2 \in P \cup T$.

- $x_1 \leq x_2$ *(x_1 precede x_2): x_1 precede x_2 se (x_1, x_2) pertence ao fechamento reflexivo transitivo de F (F^*), ou seja, se existe um caminho entre x_1 e x_2.*

- $x_1 \# x_2$ *(x_1 e x_2 estão em conflito): x_1 e x_2 estão em conflito se existirem transições distintas $t_1, t_2 \in T$ tal que ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$, $t_1 \leq x_1$ e $t_2 \leq x_2$*

- $x_1 \,||\, x_2$ *(x_1 e x_2 são concorrentes): x_1 e x_2 são concorrentes se não são precedentes ($\neg\, x_1 \leq x_2$ e $\neg\, x_2 \leq x_1$) e não estão em conflito ($\neg(\, x_1 \# x_2)$).*



(a) Rede (b) Grafo de alcançabilidade

Figura 2.5: Exemplo de uma rede de Petri e seu grafo de alcançabilidade

Alguns exemplos de relações de ordem na rede da figura 2.5 são:

$$t_2 \,||\, t_6 \;,\; t_2 \# t_4 \;,\; p_1 \,||\, p_6$$

Definição 8 (Rede de ocorrência)**.** *Uma rede Condição/Evento $N = (P, T, F, W, M)$ é uma rede de ocorrência se, e somente se:*

(i) $\forall p \in P, |{}^\bullet p| \leq 1$ *(os lugares têm no máximo uma transição de entrada)*

(ii) F é acíclico (o fechamento transitivo é irreflexivo)

(iii) Nenhum evento $t \in T$ está em auto-conflito (um nó $x \in P \cup T$ está em auto-conflito se $x \# x$)

Definição 9 (Processos de ramificação). *Processos de ramificação, ou simplesmente processos, são rotulamentos de uma rede que representam todos os caminhos possíveis mantendo as informações sobre concorrência e conflitos. O processo de ramificação de uma rede N é o par $\beta = (N, \varphi)$ onde $N = (P, T, F, W, M)$ é uma rede de ocorrência e φ é uma função de rotulamento que satisfaz as seguintes propriedades:*

(i) $\varphi(P) \subseteq P$ *e* $\varphi(T) \subseteq T$ *(preserva a natureza dos nós)*

(ii) $\forall t \in T$, *a restrição de* φ *para* ${}^\bullet t$ *é uma função bijetora entre* ${}^\bullet t$ *(em* N*) e* ${}^\bullet\varphi(t)$ *(em* β*), e simetricamente para* $t^\bullet$ *e* $\varphi(t)^\bullet$ *(preserva o ambiente das transições)*

(iii) Sendo $min(N)$ *o conjunto dos elementos mínimos de* $P \cup T$ *que possuem seu pré-conjunto vazio, a restrição de* φ *para* $min(N)$ *é uma função bijetora entre* $min(N)$ *e* M_0 *(*β *começa pela marcação inicial). Considerando redes onde não existem transições com pré-conjunto vazio,* $min(N)$ *só pode conter condições (*P*), portando* $min(N) = M_0$.

(iv) $\forall t_1, t_2 \in T$, *se* ${}^\bullet t_1 = {}^\bullet t_2$ *e* $\varphi(t_1) = \varphi(t_2)$ *então* $t_1 = t_2$ *(*β *não duplica as transições de* N*)*

Embora o processo de ramificação possa ser infinito, é possível truncá-lo em uma subrede denominada *prefixo finito*, que possui todas as marcações alcançáveis. Esse prefixo é o desdobramento, mas para formalizar sua definição ainda são necessários os conceitos de configuração, configuração local, prefixo finito completo e evento de corte. A figura 2.6 é o desdobramento da rede exemplo.

Definição 10 (Configuração). *Uma configuração* C *é um conjunto de transições que satisfazem as seguintes condições:*

(i) $\forall e' \leq e,\ e \in C \Rightarrow e' \in C$

(ii) $\forall e_1, e_2 \in C,\ e_1 \neq e_2 \Rightarrow {}^\bullet e_1 \cap {}^\bullet e_2 = \emptyset$

Uma configuração de uma rede N pode ser associada com uma marcação $mark(C)$ que corresponde a uma marcação alcançável a partir de M_0 após todas as transições de

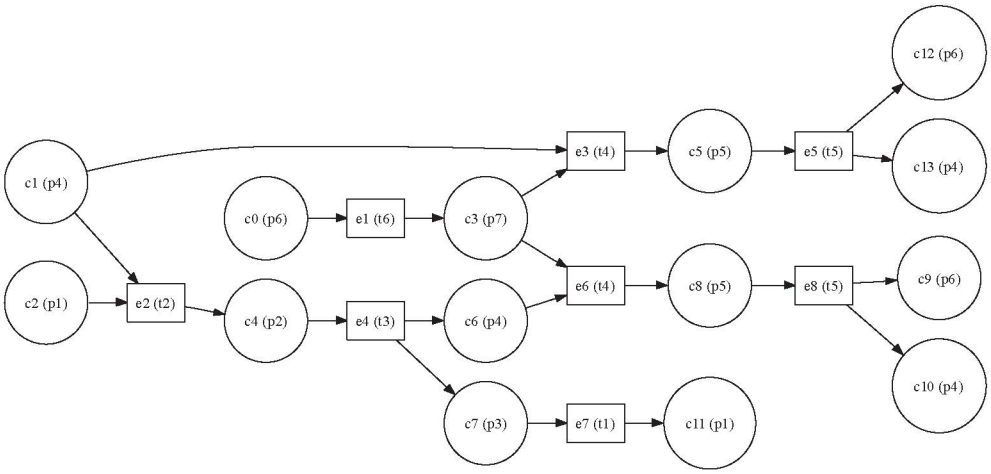


Figura 2.6: Desdobramento da rede da figura 2.5

C terem sido disparadas.

$$mark(C) = (min(N) \cup C^{\bullet}) \backslash {}^{\bullet}C$$

Na figura 2.6, a seqüência de transições/eventos $\{e_2, e_4, e_1, e_6\}$ é uma configuração e a marcação alcançável a partir dessa configuração é $\varphi(\{c_7, c_8\}) = \{p_3, p_5\}$.

Definição 11 (Configuração local). *A configuração local de um evento e, denotada por* $[e]$*, é a configuração mínima que contém e e todos os seus precedentes.*

Por exemplo, a configuração mínima para o evento e_8 é $[e_8] = \{e_2, e_4, e_1, e_6, e_8\}$

Na maioria dos casos, o desdobramento β da rede é infinito, mas é possível truncá-lo em uma subrede denominada prefixo finito completo β', que possui todas as informações de β.

Definição 12 (Prefixo Finito Completo). *O prefixo β' de β é completo se para cada marcação alcançável M, existe uma configuração $C \in \beta'$ tal que:*

1. $\mathtt{Mark}(C) = M$
2. *para cada transição t habilitada por M, existe uma configuração $C \cup \{e\}$ tal que $e \notin C$ e $\varphi(e) = t$*

A rede da figura 2.6 é um prefixo finito completo que foi cortado do desdobramento sem

perder informações. Esse corte é feito identificando eventos que levam a uma marcação já representada, denominados eventos de corte (*cut-off*).

Definição 13 (Evento de corte). *Seja $\prec$ uma ordem adequada (adequate order) nas configurações do desdobramento e seja β o seu prefixo contendo um evento e. O evento e é um evento de corte (cut-off) de β se β contém a configuração local $[e']$ tal que:*

(i) $\mathtt{Mark}([e]) = \mathtt{Mark}([e'])$

(ii) $[e'] \prec [e]$

Quando $[e'] < [e]$, pode-se dizer que $[e']$ é menor que $[e]$ se ele tiver menos eventos, ou seja, $|[e']| < |[e]|$. Como isso é fácil verificar que essa ordem parcial é uma ordem adequada, portanto $[e'] \prec [e]$.

A construção do prefixo finito completo é feito seguindo o algoritmo melhorado de Esparza *et al* [4], também chamado de algoritmo Desdobramento ERV (*ERV Unfolding*). O algoritmo é apresentado a seguir.

Algoritmo 1 Construção do prefixo finito completo

entrada: Uma rede N, onde $M_0 = \{p_1, \cdots, p_k\}$
saída: Um prefixo finito completo do desdobramento`Unf`
`Unf` $\leftarrow$ lugares de M_0
`pe` $\leftarrow$ transições habilitadas por M_0
`corte` $\leftarrow \emptyset$
while `pe` $\neq \emptyset$ **do**
  escolha um evento $e = (t, X)$ de `pe` tal que $[e]$ seja mínima respeitando a ordem $\prec$
  if $[e] \cap$ `cut-off` $= \emptyset$ **then**
    adicione e e seu pós-conjunto a `Unf`
    `pe` $\leftarrow$ `PE(Unf)` {Atualiza as transições habilitadas}
    if e é um evento de corte **then**
      `corte` $\leftarrow$ `cut-off` $\cup\, e$
    end if
  else
    `pe` $\leftarrow$ `pe` $\setminus \{e\}$
  end if
end while

2.5.1 Ferramenta *MOLE*

Existem vários desdobradores disponíveis ao acesso público, tais como o *PUNF* e o *ERVunfold*. O desdobrador *MOLE* foi utilizado neste projeto por ter sido o desdobrador utilizado pelo *Petrify* (seção 3.7).

O desdobrador *MOLE*[1] é uma ferramenta *freeware* para gerar desdobramentos de redes de Petri, com a limitação de operar apenas com redes seguras.

Foi desenvolvida para ser compatível com o ambiente do projeto PEP (*Programming Environment based on Petri Nets*) mantido pelo grupo de Sistemas Paralelos do Departamento de Ciência da Computação da Universidade de Oldenburg na Alemanha.

A entrada do programa é um arquivo com a descrição da rede no formato próprio do ambiente e a saída é o desdobramento em um arquivo no formato também utilizado pelas ferramentas do ambiente PEP. Anexo à ferramenta há um aplicativo para conversão dos arquivos de saída para formatos gráficos.

O prefixo mostrado na figura 2.6 é o prefixo finito completo gerado pelo *MOLE* para a rede da figura 2.5.

2.6 Considerações

As redes de Petri são utilizadas como ferramenta para resolver problemas de planejamento desde a década de 80. O desempenho de planejadores baseados em RdP não supera a eficiência de outras abordagens, mas tem sido um campo de diversas pesquisas e resultados.

Em 2000, Silva *et al* [17] fizeram uma proposta de transformação do grafo de planos em uma rede de Petri acíclica e trataram o problema de planejamento como um problema de alcançabilidade.

A partir desse trabalho, foram derivados vários outros dentro do grupo de pesquisa em planejamento em inteligência artificial do Laboratório de Inteligência Artificial e Metodos Formais da UFPR. O próprio algoritmo do Petriplan teve cinco versões, uma delas resul-

[1] http://www.fmi.uni-stuttgart.de/szs/tools/mole/

tando em um novo trabalho de Silva, a Rede de Planos [15], que abandona a estrutura de grafo de planos provinda do *Graphplan* e traduz os problemas de planejamento em PDDL diretamente em uma rede de Petri acíclica.

Montaño [18] apresenta uma técnica onde o problema de alcançabilidade é tratado por um resolvedor SAT baseado em fórmulas não-clausais, a partir do seu mapeamento para fórmulas em NNF (*Negation Normal Form*). Carvalho [19] desenvolveu o GAPNet (*Genetic Algorithm for Reachability on Petri Nets*), que propõe uma solução para o problema de alcançabilidade utilizando o paradigma dos algoritmos genéticos.

Como será visto no capítulo a seguir, avanços na resolução de alcançabilidade redes de Petri podem ser utilizados para a resolução de problemas de planejamento. Neste trabalho, será mostrado como o algoritmo de desdobramento de redes de Petri pode ter resultados de mesmo nível que soluções clássicas para planejamento.

CAPÍTULO 3

PLANEJAMENTO

Neste capítulo são apresentados os conceitos relativos a planejamento em Inteligência Artificial que serão utilizados no trabalho. São descritos conceitos necessários para a compreensão do algoritmo *Petrigraph* proposto, assim como outros algoritmos de resolução de problemas de planejamento. Destes algoritmos, detalhamos o algoritmo *Petrify*, que forma a base para o algoritmo proposto no estudo.

3.1 Definição de Planejamento

A definição de planejamento utilizada neste trabalho é a de planejamento clássico. O problema de planejamento clássico em Inteligência Artifical pode ser definido como, dados um estado inicial e um estado final de um sistema, encontrar uma seqüência de ações chamada *plano*, que leve o sistema do estado inicial ao final.

Definição 14 (Literais). *Define-se V como um conjunto de variáveis de estado booleanas. O conjunto de literais sobre este conjunto V é $L = V \cup \{\neg v | v \in V\}$, onde o complemento $\overline{l}$ de um literal $l \in L$ é definido por $\overline{v} = \neg v$ e $\overline{\neg v} = v$ para $v \in VA$.*

Literais representam os possíveis estados das variáveis do problema de planejamento, e são utilizados nas pré-condições e pós-condições dos operadores deste problema.

Definição 15 (Operadores de planejamento). *Um operador de planejamento é uma dupla $o = \langle P, E \rangle$, onde P é um conjunto de literais de pré-condição e E é um conjunto de literais de pós-condição ou efeito.*

Um operador de planejamento representa um possível conjunto de alterações nas variáveis de estado que definem as literais.

Definição 16 (Problemas de planejamento). *Um problema de planejamento é uma quádrupla $\langle V, I, O, G \rangle$ onde V é um conjunto de variáveis de estado, $I : V \rightarrow \{0, 1\}$ é um*

estado inicial, O é um conjunto de operadores de planejamento, e G é um conjunto de literais de objetivo.

3.2 Algoritmos de planejamento

São descritos a seguir alguns algoritmos de resolução de problemas de planejamento relacionados com o estudo feito. Estes algoritmos resolvem o planejamento convertendo problemas para outras estruturas tratáveis.

Graphplan [20] baseia-se na construção de um grafo de planos (*plan graph*), um grafo direcionado onde todos os operadores possíveis para cada passo do plano estão expressos. O grafo de dois níveis (estados e ações) é criado somente com os literais existentes na posição seqüencial 0. Em cada passo i do algoritmo, o grafo do passo $i-1$ é estendido seqüencialmente em um passo, os *mutexes* (definidos a seguir) são marcados, e uma busca por um plano válido de comprimento i é feita no grafo resultante, caso as literais do objetivo estejam presentes na camada i.

O Graphplan inovou por ser capaz de representar *mutexes* em sua estrutura. Um mutex representa a impossibilidade de dois operadores de planejamento serem executados simultaneamente ou de duas variáveis de estado estarem ativas simultaneamente. A representação destes mutexes no grafo de planos restringe o espaço de busca, reduzindo o tempo necessário para resolução.

Trabalhando a partir do Graphplan, Silva *et al* [17] criaram o Petriplan, um algoritmo de conversão de um problema de planejamento para uma rede de Petri acíclica. Ele utiliza o Graphplan como passo intermediário, e dessa forma produz uma rede de Petri dividida em camadas - mas, ao contrário do Graphplan, os lugares de cada camada não estão ordenados de forma estritamente seqüencial: operadores podem vir a ser ativados antes de outros operadores que estejam em camadas anteriores da rede.

O Petriplan aproveita algoritmos utilizados para resolver problemas de alcançabilidade em redes de Petri, mas tem a desvantagem de não ter sido criado por um método totalmente apropriado para aproveitar estes algoritmos ao máximo. Teoricamente, um processo de conversão de problema de planejamento diretamente para rede de Petri irá manter in-

formações relativas à estrutura do problema que poderiam ser usadas para a otimização do algoritmo de resolução, tais como similaridades entre variáveis de estado e operadores.

Uma área com rápido desenvolvimento recente é a de utilização de técnicas para problemas de SAT, ILP e CSP em problemas de planejamento. Satplan, proposto por Bartz and Seldman [21] tenta resolver o problema de planejamento convertendo-o para um problema de satisfabilidade booleana, ou SAT. Predicados inicias, predicados finais e ações são todos convertidos para axiomas lógicos, e tenta-se encontrar um modelo que satisfaça estes axiomas. Este tipo de representação é bastante apropriado à inclusão de limitações extras posicionadas temporalmente - como a exigência que uma ação seja ou não executada no intervalo temporal n.

Avanços foram feitos no controle heurístico de planejadores, de forma a encontrar um plano mais rapidamente. Em particular, heurísticas podem ser customizadas para o domínio específico no qual os problemas serão resolvidos.

Adicionalmente, o planejamento clássico está sendo estendido para cobrir uma maior gama de problemas. Extensões a problemas de planejamento incluem:

- Eventos durativos, ou seja, que levam um certo período de tempo para finalizarem, e cuja ordem deve ser otimizada em situações de concorrência;
- Problemas estocásticos, onde eventos não tem garantia de produzir suas pós-condições, exceto como uma condição probabilística;
- Domínios dinâmicos, onde condições podem mudar independentemente do uso de ações no problema;
- Sistemas com conhecimento imperfeito do estado corrente;
- Problemas nos quais a satisfação parcial do objetivo é possível.

3.3 PDDL

PDDL (*Planning Domain Definition Language)* é uma linguagem desenvolvida por McDermott *et al* [7] para a descrição de problemas de planejamento, criada como uma

evolução das linguagens STRIPS [22] e ADL [23] para a competição de planejamento AIPS-98.

PDDL divide a formulação do problema em duas partes: o *domínio PDDL* e o *problema PDDL*. O domínio PDDL especifica *tipos* de *objetos*, uma lista de *predicados* existentes no problema e uma lista de ações existentes.

Definição 17 (Objeto). *Objetos são entidades de um problema de planejamento definidas de maneira virtual. Objetos podem pertencer a conjuntos denominados tipos.*

Um objeto não tem existência prática no problema além dos predicados definidos sobre ele.

Definição 18 (Predicado). *Um predicado* $pred(a_1, \ldots, a_n)$ *é composto por um nome pred e uma lista de argumentos* $(a_1, \ldots, a_n)$*, cada um dos quais pode ter um tipo de objeto associado. Um predicado em PDDL define uma classe de variáveis de estado no problema de planejamento, com uma variável para cada combinação de valores possível para os argumentos do predicado.*

Assim, um predicado definido como `sobre(x:bloco y:bloco)` em um domínio PDDL, dado um problema com três objetos de tipo `bloco`, chamados `a`, `b` e `c`, este predicado define uma classe de variáveis de estado contendo `sobre(a a)`, `sobre(a b)`, ..., `sobre(c b)`, `sobre(c c)`.

Definição 19 (Ação). *Uma ação* $ac(a_1, \ldots, a_n, pre, pos, del)$ *é composta por um nome ac, uma lista de argumentos* $(a_1, \ldots, a_n)$*, cada um dos quais pode ter um tipo de objeto associado, e conjuntos de predicados de pré-condição (pre), pós-condição de ativação (pos) e pós-condição de remoção (del).*

Uma ação no domínio de planejamento define uma classe de operadores no problema de planejamento, com um operador para cada combinação de valores possível para os argumentos da ação.

Por exemplo, uma ação `pegar` ($a_1 \in t_{bloco}$, $a_2 \in t_{bloco}$, $pre = del = \{$`vazia()`, `livre`(a_1), `sobre`$(a_1, a_2)\}$, $pos = \{$`garra`(a_1), `livre`$(a_2)\}$), com os mesmos três objetos de tipo `bloco`

no exemplo anterior, irá definir uma classe de operadores contendo `pegar(a a)`, `pegar(a b)`, ..., `pegar(c b)`, `pegar(c c)`. Cada um destes operadores terá pré-condições e efeitos apropriados: O operador `pegar(a b)`, por exemplo, terá $o_{\texttt{pegar(ab)}} = \langle P = \{$ `vazia ()`, `livre (a)`, `sobre (a b)` $\}, E = \{$ `garra (a)` , `livre (b)`, $\neg$`vazia ()`, $\neg$ `livre (a)`, $\neg$ `sobre (a b)` $\}\rangle$.

```
(define (domain BLOCKS)
  (:requirements :strips :typing)
  (:types bloco)
  (:predicates (sobre ?a - bloco ?b - bloco)
               (mesa ?a - bloco)
               (livre ?a - bloco)
               (vazia)
               (garra ?a - bloco)
               )
  (:action pegar_mesa
             :parameters (?a - bloco)
             :precondition (and (livre ?a) (mesa ?a) (vazia))
             :effect
             (and (not (mesa ?a))
                  (not (livre ?a))
                  (not (vazia))
                  (garra ?a)))
  (:action depositar_mesa
             :parameters (?a - bloco)
             :precondition (garra ?a)
             :effect
             (and (not (garra ?a))
                  (livre ?a)
                  (vazia)
                  (mesa ?a)))
  (:action pegar
             :parameters (?a - bloco ?b - bloco)
             :precondition (and (sobre ?a ?b) (livre ?a) (vazia))
             :effect
             (and (garra ?a)
                  (livre ?b)
                  (not (livre ?a))
                  (not (vazia))
                  (not (sobre ?a ?b))))
  (:action depositar
             :parameters (?a - bloco ?b - bloco)
             :precondition (and (garra ?a) (livre ?b))
             :effect
             (and (not (garra ?a))
                  (not (livre ?b))
                  (livre ?a)
                  (vazia)
                  (sobre ?a ?b)))
```

Figura 3.1: Domínio BlocksWorld descrito em PDDL

```
(define (problem blocks-3-0)
(:domain blocks)
(:objects a b c - bloco)
(:init (livre c) (livre a) (livre b) (mesa c) (mesa a) (mesa b) (vazia))
(:goal (and (sobre c b) (sobre b a)))
)
```

Figura 3.2: Problema BlocksWorld com três blocos descrito em PDDL

O *problema PDDL* é um documento que define os objetos existentes e seus tipos, as variáveis de estado ativadas no estado inicial do sistema e os literais de objetivo. Um exemplo de definição de domínio e problema PDDL pode ser visto nas figuras 3.1 e 3.2.

3.4 Petrify

Petrify é o algoritmo usado por Hickmott *et al* [1] para a construção, a partir de um problema de planejamento, de uma rede segura que possa passar pelo desdobramento no *MOLE* (seção 2.5.1). Como demonstrado por Silva [15], o problema de alcançabilidade em planejamento é equivalente ao problema de alcançabilidade em uma rede de Petri. Para que a rede resultante seja segura (definição 4) conforme requerido pelo algoritmo de desdobramento, existem as seguintes ressalvas:

- Os operadores devem ser seguros, ou seja, devem desativar o mesmo número de variáveis de estados que ativam, para que a rede criada também seja segura.
- Operadores não podem ter pré-condições negativas, ou seja, dependentes da ausência de um predicado no sistema.

O algoritmo *Petrify* descreve algoritmos para alterar os operadores de forma que eles cumpram estas restrições. Para que os operadores sejam seguros, eles não podem afetar pós-condições que não sejam pré-condições daquele operador, ou seja, todas as variáveis de estado desativadas por um operador devem ser pré-condições daquele operador. No planejamento, isso é equivalente a limitar operadores a alterar o valor de variáveis de estado que estejam incluídas na lista de pré-condições dos mesmos.

O *Petrify* resolve esse impasse com o conhecimento que as variáveis de estado só podem ter dois valores. Assim, cada operador de planejamento $o = \langle p, e \rangle$, sendo e o conjunto de literais representadas em e mas não em p, é substituído por $2^{|e'|}$ operadores $o_n = \langle p_n, e_n \rangle$, de forma que todas as combinações de $\{l \wedge \neg l | l \in e'\}$ estejam expressas.

Por exemplo, para um operador $o = \langle p = \{\mathsf{a}, \neg\mathsf{b}, \mathsf{c}\}, e = \{\neg\mathsf{a}, \mathsf{b}, \mathsf{d}, \neg\mathsf{e}\} \rangle$, $e' = \{\mathsf{d}, \neg\mathsf{e}\}$. O operador é substituído por quatro operadores seguros $o_i = \langle p_i, e_i \rangle$, onde:

$$
\begin{aligned}
p_1 &= \{\mathsf{a}, \neg\mathsf{b}, \mathsf{c}, \mathsf{d}, \neg\mathsf{e}\} & e_1 &= \{\neg\mathsf{a}, \mathsf{b}\} \\
p_2 &= \{\mathsf{a}, \neg\mathsf{b}, \mathsf{c}, \neg\mathsf{d}, \neg\mathsf{e}\} & e_2 &= \{\neg\mathsf{a}, \mathsf{b}, \mathsf{d}\} \\
p_3 &= \{\mathsf{a}, \neg\mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}\} & e_3 &= \{\neg\mathsf{a}, \mathsf{b}, \neg\mathsf{e}\} \\
p_4 &= \{\mathsf{a}, \neg\mathsf{b}, \mathsf{c}, \neg\mathsf{d}, \mathsf{e}\} & e_4 &= \{\neg\mathsf{a}, \mathsf{b}, \mathsf{d}, \neg\mathsf{e}\}
\end{aligned}
$$

A questão de pré-condições negativas é resolvida substituindo-se cada variável de estado v por duas variáveis de estado, v e $\hat{v}$, e os operadores são refeitos de forma que a pós-condição $\{v\}$ é substituída por $\{v, \neg\hat{v}\}$ e a pós-condição $\{\neg v\}$ é substituída por $\{\neg v, \hat{v}\}$. Efetivamente, cada variável de estado é substituída por um par de lugares na rede resultante, onde qualquer das transições que consuma uma marca de um dos lugares do par adicionará uma marca ao outro.

Por exemplo, no domínio ROVERS, que descreve veículos móveis que podem comunicar dados sobre o terreno a seu redor para uma base de comunicações, a variável de estado `dt(x)` define se os dados de terreno referentes à localidade `x` foram comunicados à base. O operador `comunicar_dt(x, ...)` adiciona `dt(x)`, mas não verifica se o mesmo predicado existia ou não anteriormente à operação. Assim, o operador é substituído pelo par {`comunicar_dt'(x, ...)`,`comunicar_dt''(x, ...)`} onde `comunicar_dt'(x, ...)` tem pré-condição $\widehat{\texttt{dt(x)}}$ e efeito $\neg\widehat{\texttt{dt(x)}}$, `dt(x)`, e `comunicar_dt''(x, ...)` tem pré-condição `dt(x)` e nenhum efeito relacionado a `dt(x)`.

A figura 3.3 mostra uma rede de Petri gerada para o operador $x = \langle\{\texttt{a}, \neg\texttt{b}\}, \{\neg\texttt{a}, \texttt{c}\}\rangle$, após a decomposição em dois operadores seguros com pré-condições positivas $x'_1 = \langle\{\texttt{a}, \hat{\texttt{b}}, \texttt{c}\}, \{\neg\texttt{a}, \hat{\texttt{a}}\}\rangle$ e $x'_1 = \langle\{\texttt{a}, \hat{\texttt{b}}, \hat{\texttt{c}}\}, \{\neg\texttt{a}, \hat{\texttt{a}}, \neg\hat{\texttt{c}}, \texttt{c}\}\rangle$.

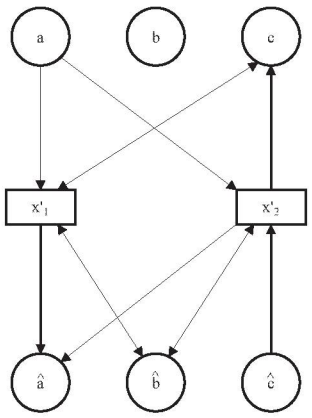


Figura 3.3: Rede de Petri gerada para um operador do *Petrify*

A rede gerada pelo *Petrify* tem $2 \cdot (|A|)$ lugares e $\sum^{i} 2^{v(O_i)}$ transições, onde A é o conjunto de variáveis de estado do problema e $v(O_i)$ é o número de variáveis de estado que são afetadas pelo operador O_i mas não estão incluídos na lista de pré-condições deste, conforme o algoritmo descrito acima.

O algoritmo *Petrify* tem a vantagem de não precisar da definição de operadores mutex na rede criada: como efeito colateral da semântica da rede de Petri, a análise de ações possíveis durante o desdobramento automaticamente considera operadores de exclusão. Além disso, o desdobramento duplica nós de forma a garantir que nós de condições (variáveis de estado) tenham um único nó de evento (operador) os precedendo. Assim, uma vez encontrada uma marcação de objetivo, os passos do plano podem ser encontrados fazendo uma busca no prefixo a partir de cada lugar marcado em direção à raiz, sem ser necessário monitorar as operações tomadas durante o algoritmo de resolução do problema de alcançabilidade.

Uma análise do *Petrify* revela a falta de otimização da rede Condição/Evento. A rede Condição/Evento resultante sempre terá um número de condições igual ao dobro de variáveis de estado do problema de planejamento. Uma vez que o tamanho do prefixo finito completo no pior caso é $O(2^n)$, onde n é o tamanho da rede de desdobramento, uma diminuição de tamanho da rede de Petri nesse pior caso terá efeito exponencial no tempo de execução do algoritmo de desdobramento. A seguir, serão mostrados algoritmos de compressão do espaço de estados de planejamento que também podem ser utilizados na criação de redes Condição/Evento, e que serão utilizados no *Petrigraph* (capítulo 5).

3.5 Variáveis multivaloradas

Na definição de problema de planejamento, as variáveis de estado são *booleanas*: Elas podem ter apenas os valores 0 e 1, refletidas em PDDL pela definição do predicado equivalente. Assim, o armazenamento dessas variáveis em memória requer 1 bit por variável de estado, o que é simples para problemas pequenos, mas torna-se rapidamente mais complicado com o aumento de predicados.

Por exemplo, no domínio BlocksWorld, cada bloco pode estar posicionado sobre uma mesa ou sobre qualquer um dos outros blocos. Assim, o número de variáveis de estado associadas à posição dos blocos, com n blocos, é igual a n^2. Um problema com 100 blocos terá que manter 10000 variáveis de estado.

Edelkamp e Helmert [5] propõem a compressão desse armazenamento. O algoritmo

proposto por eles perfaz uma análise do problema PDDL, inicialmente procurando predicados que não precisem ser representados por variáveis de estado: predicados *constantes* e de *caminho único.*

Definição 20 (Predicados constantes). *Predicados que não são removidos por nenhuma ação do domínio.*

Estes predicados não precisam ser representados por variáveis de estado.

Definição 21 (Predicados de caminho único). *Predicados que são removidos por ações do domínio, mas não são adicionados por nenhuma ação.*

Estes predicados somente precisam ser representados por variáveis de estado se forem parte do estado inicial do problema de PDDL.

Predicados constantes e de caminho único são encontrados através da busca dos predicados de remoção e pós-condição das ações do domínio, um processo de complexidade $O(a)$, onde a é o número de ações definidas no domínio PDDL. No passo seguinte, o algoritmo passa a fazer a análise dos predicados para poder exprimí-los em variáveis de estado multivaloradas.

Definição 22 (Variável Multivalorada). *Uma variável de estado que pode ter valores em* $\{1 \ldots x\}$, *onde* $x \in \mathbb{N}$ *e* $x > 0$.

O processo inicia-se com a análise dos predicados alcançáveis no problema. Isso é feito efetuando-se uma busca no espaço de estados:

1. Definimos o grupo G como o grupo de variáveis de estado definidas inicialmente no problema.

2. Para cada item em G, se houver um operador no domínio que tenha G como pré-condição, adicione todas as pós-condições desse operador a G. Repita esse passo até não existirem mais variáveis de estado a serem adicionadas.

O número de comparações da busca é de $O(v)$ no melhor caso e $O(v^2)$ no pior caso, onde v é o número de variáveis de estado do problema. Nesse ponto, temos todas as variáveis

de estado que podem ser utilizadas na resolução do problema. Procedemos à análise dos predicados do domínio PDDL, a fim de confirmar se estes estão *balanceados*, como definido a seguir, ou em caso contrário, efetuar procedimentos para torná-los balanceados.

Definição 23 (Predicado balanceado). *Um predicado* `pred` *com n argumentos é balanceado em relação a* $i \leq n$ *se toda ação que remover um predicado* `pred` $(p_1 \ldots p_i \ldots p_n)$ *adicionar um, e apenas um, predicado* `pred` $(p_1 \ldots p'_i \ldots p_n)$.

Se um predicado é balanceado com relação a i, e tomando $npred_i(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n)$ como o número de objetos p_i para os quais o predicado `pred` $(p_1, \ldots, p_n)$ é verdadeiro, temos a seguinte situação:

- $npred_i(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n) = 0$: Os predicados `pred` $(p_1, \ldots, p_n)$ sao constantes (definição 20) e nenhum precisa ser armazenado como variável de estado.

- $npred_i(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n) = 1$: O estado dos predicados `pred` $(p_1, \ldots, p_n)$ pode ser representado por uma única variável de estado multivalorada com n valores, onde n é o número de objetos p_i para os quais o predicado `pred` $(p_1, \ldots, p_n)$ é possível.

- $npred_i(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n) > 1$: Os predicados não podem ser representados de forma reduzida, pelo menos em relação a i, uma vez que uma representação por via de variável de estado multivalorada exigiria que essa variável armazenasse dois ou mais valores diferentes simultaneamente.

Às vezes, um predicado pode não ser balanceado, mas ações que removem um predicado `a` sempre adicionam um predicado `b`, e vice-versa. É efetuado um teste recursivo de *combinação de predicados* para resolver esta situação. O processo recursivo de combinação funciona da seguinte maneira: Para cada predicado `pred` e cada argumento i deste predicado:

1. Verifica-se se o predicado `pred` é balanceado em relação a i. Se sim, o algoritmo retorna o conjunto $\langle$`pred`$, i\rangle$.

2. Se o predicado não for balanceado, verifica-se todas as ações que removam o predicado `pred` com argumentos $(p_1, \ldots, p_n)$ e uma busca é feita nas pós-condições destas ações por:

   - Predicados `pred'` com argumentos $(p_1, \ldots, p_{i-1}, p'_i, p_{i+1}, \ldots, p_n)$, onde p'_i é um argumento qualquer;
   - Predicados `pred'` com argumentos $(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n)$.

3. Para cada predicado encontrado dessa maneira, pode-se efetuar a combinação de predicados da seguinte forma: Um predicado `pred+pred'` é criado com argumentos $(p_1,\ldots,p_{i-1},p''_i,p_{i+1},\ldots,p_n)$, onde p''_i é um novo argumento cuja construção é descrita no item a seguir, e todas as ocorrências de `pred` e `pred'` no domínio são substituídas pelo novo predicado.

4. Caso o `pred'` encontrado tenha um argumento p'_i, os valores possíveis para o argumento p''_i são tanto os de p_i quanto os de p'_i, sem que haja interseção entre os dois, ou seja, se o mesmo valor v for possível para p_i e p'_i, o argumento p''_i terá um valor v para $p_i = v$ e v' para $p'_i = v$. Caso o `pred'` encontrado tenha um argumento a menos que `pred`, p''_i pode assumir todos os valores de p_i, mais um novo valor $v_{pred'}$, que será utilizado nos casos em que `pred+pred'` substituir `pred'`.

5. Para cada predicado criado dessa maneira, substitui-se o predicado criado no domínio e o processo é refeito.

O resultado deste algoritmo é o conjunto de predicados balanceados possíveis no problema. Cada predicado balanceado em relação a i é representado por uma única variável de estado multivalorada, com valores entre 0 e n, onde n é o número de objetos que podem ser definidos como o argumento p_i do predicado. A complexidade desta etapa, relativa ao número de domínios formados por combinações de predicados que precisam ser examinados, depende do número de predicados que podem ser combinados. A complexidade é de $O(2^p n_p)$ no pior caso, onde p é o número de predicados do domínio e n_p é a soma do número de argumentos destes predicados.

Assim, a complexidade total do algoritmo é de $O(2^p n_p + v^2)$ no pior caso. Qual dos elementos é dominante depende da modelagem específica do domínio e do tamanho do problema. Em particular, o valor $O(2^p n_p)$ permanece invariante enquanto o domínio for mantido, independentemente do tamanho do problema.

Como exemplo, é descrito a seguir o processo operando no domínio BLOCKSWORLD com o conjunto de blocos B. O domínio descreve um conjunto de blocos, cada um dos quais pode estar diretamente sobre uma mesa ou empilhado sobre outro bloco, e uma garra capaz de segurar um bloco por vez e reposicioná-lo.

O domínio contém os seguintes predicados:

- `sobre` ($p_1 \in B, p_2 \in B$): O bloco p_1 está sobre o bloco p_2.
- `mesa` ($p_1 \in B$): O bloco p_1 está diretamente sobre a mesa.
- `livre` ($p_1 \in B$): O bloco p_1 não tem blocos sobre ele.
- `garra` ($p_1 \in B$): O bloco p_1 está seguro pela garra.
- `vazia` (): A garra não está segurando nenhum bloco.

E as seguintes ações:

- `pegar` ($a_1 \in B, a_2 \in B, pre = del =$ {`vazia`(),`livre`(a_1),`sobre`(a_1, a_2)}, $pos =$ {`garra`(a_1),`livre`(a_2)}): A garra adquire o bloco a_1, que está sobre o bloco a_2.
- `pegar_mesa` ($a_1 \in B$, $pre = del =$ {`vazia` (), `livre` (a_1), `mesa` (a_1)}, $pos =$ {`garra` (a_1)}): A garra adquire o bloco a_1, que está sobre a mesa.
- `depositar` ($a_1 \in B$, $a_2 \in B$, $pre = del =$ {`garra` (a_1), `livre` (a_2)},$pos =$ {`vazia` (), `livre` (a_1), `sobre` (a_1, a_2)}): Empilha o bloco a_1, seguro pela garra, sobre o bloco a_2.
- `depositar_mesa` ($a_1 \in B, pre = del =$ {`garra`(a_1)}, $pos =$ {`vazia`(), `livre`(a_1), `mesa`(a_1)}): Deposita o bloco a_1, seguro pela garra, sobre a mesa.

Com $B = \{a, b, c\}$, estes predicados e ações definem 19 variáveis de estado e 24 operadores.

O domínio BLOCKSWORLD não tem predicados constantes e todas as combinações de argumentos são possíveis para todos os predicados. Assim, passa-se ao balanceamento de predicados. Verificando o balanceamento de `sobre` (p_1, p_2) com relação a p_1:

- `pegar` remove um predicado `sobre` $(p_1 = a_1, p_2 = a_2)$ e não adiciona nenhum predicado `sobre` $(p'_1, p_2 = a_2)$.
- `depositar` adiciona um predicado `sobre` $(p_1 = a_1, p_2 = a_2)$ e não remove nenhum predicado `sobre` $(p'_1, p_2 = a_2)$.
- Nenhuma outra ação afeta o predicado.

O predicado `sobre` não é balanceado com relação a p_1, mas é passível de balanceamento através da combinação com outros predicados. Examinando o conjunto de pós-condições de `pegar` por predicados com a lista de argumentos (p'_1, a_2) ou (a_2), é verificado que somente o predicado `livre` (a_2) pode ser combinado com `sobre` com relação a p_1. Os dois predicados são combinados em um predicado `sobre+livre` $(p''_1 \in B + \{v_{livre}\}, p_2 \in B)$. Substitui-se as ocorrências de `sobre` (p_1, p_2) no domínio por `sobre+livre` (p_1, p_2) e as ocorrências de `livre` (p_1) por `sobre` (v_{livre}, p_a), e é refeita a análise de balanceamento com o novo domínio contendo este novo predicado.

Verificando o balanceamento de `sobre+livre` (p_1, p_2) com relação a p_1:

- `pegar` remove um predicado `sobre+livre` $(p_1 = v_{livre}, p_2 = a_1)$ e não adiciona nenhum predicado `sobre+livre` $(p'_1, p_2 = a_1)$. `pegar` também remove um predicado `sobre+livre` $(p_1 = a_1, p_2 = a_2)$ e adiciona um predicado `sobre+livre` $(p'_1 = v_{livre}, p_2 = a_2)$.
- `pegar_mesa` remove um predicado `sobre+livre` $(p_1 = v_{livre}, p_2 = a_1)$ e não adiciona nenhum predicado `sobre+livre` $(p'_1, p_2 = a_1)$.
- `depositar` adiciona um predicado `sobre+livre` $(p_1 = v_{livre}, p_2 = a_1)$ e não remove nenhum predicado `sobre+livre` $(p'_1, p_2 = a_1)$. `depositar` também remove um

predicado `sobre+livre` $(p_1 = v_{livre}, p_2 = a_2)$ e adiciona um predicado `sobre+livre` $(p'_1 = a_1, p_2 = a_2)$.

- `depositar_mesa` adiciona um predicado `sobre+livre` $(p_1 = v_{livre}, p_2 = a_1)$ e não remove nenhum predicado `sobre+livre` $(p'_1, p_2 = a_1)$.

Novamente, `sobre+livre` não é balanceado com relação a p_1, mas é passível de balanceamento através da combinação com outros predicados. Examinando o conjunto de pós-condições de `pegar` e `pegar_mesa` por predicados com a lista de argumentos (p'_1, a_1) ou (a_1), verifica-se que somente o predicado `garra` (p_1) pode ser combinado com `sobre+livre` com relação a p_1. Os dois predicados são combinados em um predicado `sobre+livre+garra` $(p''_1 \in B + \{v_{livre}, v_{garra}\}, p_2 \in B)$. Substitui-se as ocorrências de `sobre+livre` (p_a, p_b) no domínio por `sobre+livre+garra` (p_a, p_b) e as ocorrências de `garra` (p_a) por `sobre+livre+garra` (v_{garra}, p_a), e a análise de balanceamento é refeita com o novo predicado e novo domínio.

Verificando o balanceamento de `sobre+livre+garra` (p_1, p_2) com relação a p_1:

- `pegar` remove um predicado `sobre+livre+garra` $(p_1 = v_{livre}, p_2 = a_1)$ e adiciona um predicado `sobre+livre+garra` $(p'_1 = v_{garra}, p_2 = a_1)$. `pegar` também remove um predicado `sobre+livre+garra` $(p_1 = a_1, p_2 = a_2)$ e adiciona um predicado `sobre+livre+garra` $(p'_1 = v_{livre}, p_2 = a_2)$.

- `pegar_mesa` adiciona um predicado `sobre+livre+garra` $(p'_1 = v_{garra}, p_2 = a_1)$ e remove um predicado `sobre+livre+garra` $(p_1 = v_{livre}, p_2 = a_1)$.

- `depositar` adiciona um predicado `sobre+livre+garra` $(p_1 = v_{livre}, p_2 = a_1)$ e remove um predicado `sobre+livre+garra` $(p'_1 = v_{garra}, p_2 = a_1)$. `depositar` também adiciona um predicado `sobre+livre+garra` $(p'_1 = a_1, p_2 = a_2)$ e remove um predicado `sobre+livre+garra` $(p_1 = v_{livre}, p_2 = a_2)$.

- `depositar_mesa` adiciona um predicado `sobre+livre+garra` $(p_1 = v_{livre}, p_2 = a_1)$ e remove um predicado `sobre+livre+garra` $(p'_1 = v_{garra}, p_2 = a_1)$.

O predicado `sobre+livre+garra` é balanceado com relação a p_1. Assim, no domínio geral, pode-se representar as variáveis de estado `sobre` $(p_1 \in B, p_2) = \{0, 1\}$, `livre` $(p_2) = \{0, 1\}$ e `garra` $(p_2) = \{0, 1\}$, por variáveis combinadas `sobre+livre+garra` $(p_2) = B + \{v_{livre}, v_{garra}\}$.

É repetido o processo com o domínio original e o predicado `sobre` (p_1, p_2) com relação a p_2. O predicado não é balanceado, mas pode ser combinado com `garra` (p_1) para formar o predicado `sobre+garra` (p_1, p_2''). O predicado `sobre+garra` não é balanceado, mas pode ser combinado com `livre` (p_1) para formar o predicado `sobre+garra+livre` (p_1, p_2'') ou com `mesa` (p_1) para formar o predicado `sobre+garra+mesa` (p_1, p_2''). O predicado `sobre+garra+livre` não pode ser balanceado em relação a p_2, pois a ação `pegar` remove dois predicados `sobre+garra+livre` (p_1, p_2) para o mesmo p_1: `sobre+garra+livre` (a_1, a_2) e `sobre+garra+livre` (a_1, v_{livre}). No entanto, o predicado `sobre+garra+mesa` é balanceado em relação a p_2.

O processo é repetido com os outros predicados: `mesa` (p_1) com relação a p_1, `livre` (p_1) com relação a p_1 e `garra` com relação a p_1. Os dois primeiros não podem ser balanceados em qualquer combinação, mas `garra` pode ser combinado com `vazia` para formar o predicado `garra+vazia` (p_1'') com o valor adicional v_{vazia}.

Assim, três conjuntos de variáveis multivaloradas do domínio BlocksWorld são obtidos: `sobre+livre+garra` $(B) = B + \{v_{livre}, v_{garra}\}$, `sobre+garra+mesa` $(B) = B + \{v_{garra}, v_{mesa}\}$, e `garra+vazia` $() = B + \{v_{vazia}\}$. A equivalência entre as variáveis de estado booleanas do domínio e as novas variáveis multivaloradas fica definida da seguinte forma:

- `sobre` $(b, a) = 1 \Leftrightarrow$ `sobre` + `livre` + `garra`$(a) = b$;
- `livre` $(a) = 1 \Leftrightarrow$ `sobre` + `livre` + `garra`$(a) = v_{livre}$;
- `garra` $(a) = 1 \Leftrightarrow$ `sobre` + `livre` + `garra`$(a) = v_{garra}$;
- `sobre` $(a, b) = 1 \Leftrightarrow$ `sobre` + `garra` + `mesa`$(a) = b$;
- `garra` $(a) = 1 \Leftrightarrow$ `sobre` + `garra` + `mesa`$(a) = v_{garra}$;

- $\texttt{mesa}\ (a) = 1 \Leftrightarrow \texttt{sobre} + \texttt{garra} + \texttt{mesa}(a) = v_{mesa}$;
- $\texttt{garra}\ (a) = 1 \Leftrightarrow \texttt{garra} + \texttt{vazia}() = a$;
- $\texttt{vazia}\ () = 1 \Leftrightarrow \texttt{garra} + \texttt{vazia}() = v_{vazia}$.

No problema com três blocos, o número de variáveis de estado do problema foi reduzido de 19 para 7. Enquanto as variáveis de estado booleanas do problema poderiam assumir $2^{19} = 524288$ configurações, as variáveis multivaloradas podem assumir $5^3 \times 5^3 \times 4 = 62500$ configurações. O número de operadores permanece o mesmo.

A criação de variáveis multivaloradas, além de diminuir o espaço de estados, permite a criação de grafos de transição de domínio, que serão usados como passo intermediário na construção de redes de Petri a partir do problema de planejamento. Eles serão vistos a seguir.

3.6 Grafos de transição de domínio

Grafos de transição de domínio ou DTGs [6] são desenvolvidos a partir das variáveis multivaloradas. Cada valor possível da variável é colocado como um nó de um grafo, com arcos ligando os nós dependendo da existência de uma ação que possa mudar a variável de valor. O grafo pode ser construído durante a etapa de criação de variáveis sem adição de complexidade extra.

A definição de grafo de transição de domínio aqui é etiquetada diferentemente da descrita no artigo citado. Ao invés de serem etiquetados com as outras condições de mudança, os arcos são etiquetados com o nome do operador que os define.

Definição 24 (Grafos de transição de domínio). *Seja $P = \langle V, I, O, G \rangle$ um problema de planejamento com variáveis multivaloradas, e seja $v \in V$ uma variável de estado multivalorada de P. O grafo de transição de domínio de v é um grafo etiquetado e direcionado representado pelo par ordenado $DTG(v) = \langle D_v, A_v \rangle$ onde o conjunto de vértices D_v é igual ao conjunto de variáveis de estado refletidas pelos valores de v, e o conjunto de arcos $A_v = \{\langle o, p, e \rangle \,|\forall \langle P, E \rangle \in O, \forall p \in v, \forall e \in v\}$.*

Em outras palavras, o grafo de transição de domínio terá um vértice para cada variável de estado representada pela variável multivalorada, e um arco para cada operador que altere o valor desta variável.

Um exemplo de DTGs para o domínio BlocksWorld está na figura 3.4. Os grafos marcados `sobre+livre+garra(x)` e `sobre+garra+mesa(x)` representam três DTGs cada, onde x é respectivamente igual a a, b e c.

Grafos de transição de domínio podem ter pesos; nesse caso, cada transição terá um peso associado inteiro e não negativo. Exceto quando indicado, é assumido que todas as transições têm peso 1.

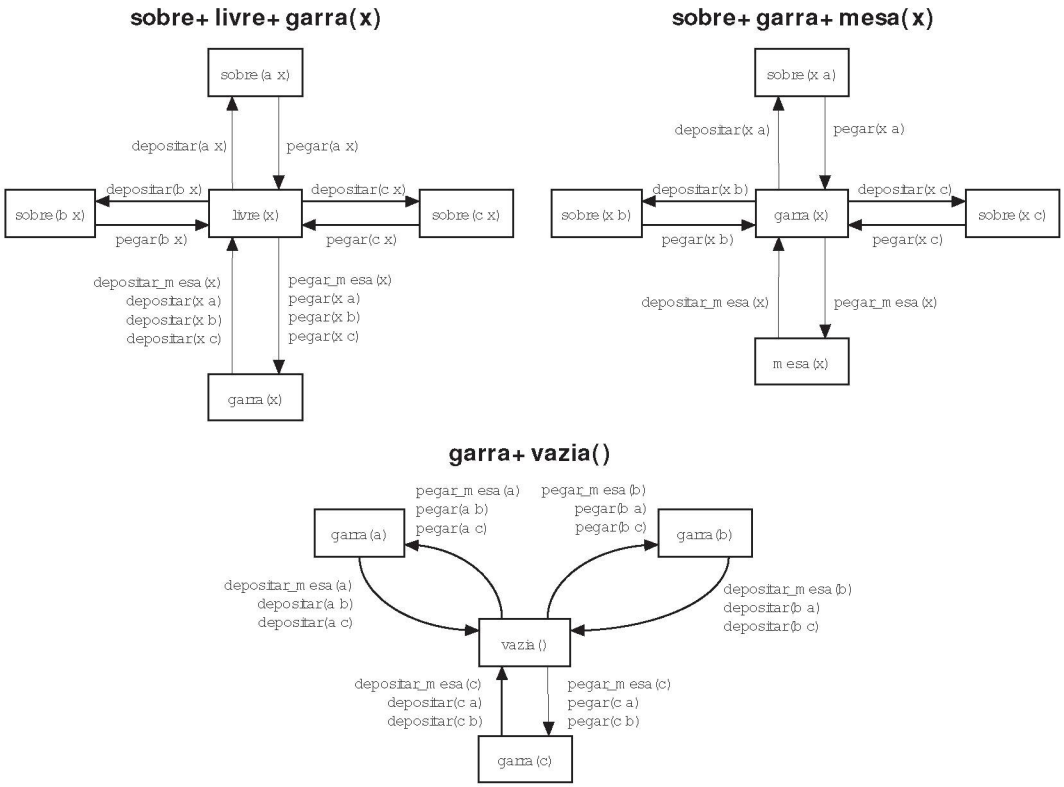


Figura 3.4: DTGs do domínio BlocksWorld com três blocos

Grafos de transição de domínio serão utilizados na construção de redes de Petri no *Petrigraph*, juntamente com o algoritmo utilizado pelo *Petrify*.

3.7 Heurísticas no desdobramento

Uma implementação de desdobramento como o *MOLE* produz o prefixo finito completo (seção 2.5) da rede de Petri gerada pelo algoritmo. Para o uso de redes de Petri em situações como problemas de planejamento, a produção do prefixo completo não é necessária se, durante o processo de desdobramento, for encontrada uma sub-rede que satisfaça as condições de alcançabilidade requeridas pelo problema de planejamento. Hickmott *et al* [1] sugerem o uso de *heurísticas* para guiar a seüuência de expansões feitas pelo algoritmo de desdobramento, de forma a otimizar essa forma de resolução. Várias heurísticas são discutidas no artigo mencionado. Para este estudo, nos concentramos na heurística h^1 [24, 25], que foi utilizada pelo *Petrify* e mostrou-se a mais simples de implementar.

Para um conjunto de literais g, uma função de custo c e um número inteiro $m \geq 1$, $h^m(s, g) =$

$$\begin{cases} 0 & \text{se } s \text{ satisfaz } g \\ \max_{\{g' \subseteq g, |g'|=m\}} h^m(s, g') & \text{se } |g| > m \\ \min_{o=\langle p,e\rangle \in O | g \cap e \neq \emptyset} c(o) + h^m(s, p) & \text{se } |g| \leq m \end{cases}$$

Ou seja, para h^1, a heurística dos lugares de objetivo será 0, e a heurística de cada lugar da rede será igual à mais baixa das somas dos valores dos lugares adjacentes a ele com a distância destes lugares. A heurística h^1 tem a vantagem de ser monotônica, ou seja, sendo $c(o)$ o custo da transição o, e $r(o, s)$ o resultado de aplicar o ao estado da rede s, $h(s) \leq h(r(o, s)) + c(o)$ para todo estado s que não seja de objetivo e toda transição o aplicável em s. A heurística h^1 não é admissível, ou seja, não garante um plano ótimo.

Para utilizar esta heurística no desdobramento, substituímos a ordem de expansões $\prec$ utilizada pelo desdobramento por uma ordem $\prec_{h^1}$ derivada de h^1. A definição de $\prec_h$ é descrita a seguir:

Definição 25 ($\prec_h$). *Sendo h uma heurística e C uma configuração, definimos $g(C) = \sum_n \in Cc(\varphi(n))$ e $f(C) = g(C) + h(mark(C))$. Definimos $C \prec_h C'$ se e somente se $f(C) < f(C') \vee (f(C) = f(C') \wedge |C| < |C'|)$.*

Em essência, cada marcação é ordenada pela soma das heurísticas de seus lugares marcados. Marcações com somas mais baixas são encaminhadas para o começo da fila de expansão.

A técnica de heurísticas aplicadas ao desdobramento, bem como as outras técnicas descritas neste capítulo, serão utilizadas em uma tentativa de construir um planejador mais eficiente que o *Petrify*. A descrição de como isto será feito procede no capítulo a seguir.

CAPÍTULO 4

PETRIGRAPH

É proposto aqui um aperfeiçoamento do método *PUP* desenvolvido por Hickmott *et al* [1], que consiste da combinação do *Petrify* com o uso de heurísticas no `MOLE`. Ao algoritmo proposto é dado o nome de *Petrigraph*.

O capítulo inicia mostrando como grafos de transição de domínio podem ser utilizados para criar redes de Petri a partir de problemas de planejamento. Em seguida é descrito o algoritmo em sua totalidade, com destaque para a comparação entre *Petrigraph* e *Petrify* e para o uso de desdobramento no *Petrigraph*. O capítulo encerra-se com a descrição da implementação do algoritmo.

4.1 Teoria

A desvantagem do *Petrify* é que o processo de criação da rede Condição/Evento não é otimizado. A rede Condição/Evento resultante sempre terá um número de condições igual ao dobro de variáveis de estado do problema de planejamento. Uma vez que o tamanho do prefixo finito completo no pior caso é $O(2^n)$, onde n é o tamanho da rede desdobrada, uma diminuição de tamanho da rede de Petri nesse pior caso terá efeito exponencial no tempo de execução do algoritmo de desdobramento.

O conceito de variáveis multivaloradas presta-se a reduzir a representação do espaço de estados de um problema de planejamento. Com o desenvolvimento de DTGs a partir destas variáveis, define-se o seguinte teorema para criar a rede de Condição/Evento a partir do problema de planejamento:

Teorema 1. *Dado um problema de planejamento* $P = \langle V, I, O, G \rangle$ *e o subconjunto de variáveis de estado* $V_d \subset V$ *o qual está associado ao conjunto de grafos de transição de domínio* $D = \{d = DTG(v) | \forall v \in V_d\}$*, é possível construir um conjunto de redes Condição/Evento para a qual toda marcação corresponda a um valor específico de uma das*

variáveis de V_d, e qualquer ativação de transições possível corresponda a uma ação válida em P para os valores de V_d definidos por estas marcações, garantindo que todas as soluções possíveis para o problema de planejamento estejam representados nesta rede.

A construção da rede Condição/Evento é feita da seguinte forma. Para um grafo de domínio $DTG(v) = \langle D_v, A_v \rangle$, é criada uma rede Condição/Evento $N_{DTG(v)} = (P, T, F, W, M)$, onde:

$$P = D_v$$
$$T = \{o | \forall \langle o, p_i, p_f \rangle \in A_v\}$$
$$F = \{\langle p_i, o \rangle, \langle o, p_f \rangle | \forall \langle o, p_i, p_f \rangle \in A_v\}$$
$$W : F \rightarrow 1$$
$$M : P \rightarrow \{0, 1\}$$

Todos os lugares da rede começam sem marcações, exceto pelo lugar correspondente à variável de estado definida em O. Pela definição de variáveis multivaloradas, sabemos que apenas uma variável de estado terá esta propriedade. Como a rede tem inicialmente uma única marcação e nenhuma transição tem mais arcos de saída do que de entrada, garante-se que a rede gerada é segura. A existência da marcação em um lugar v equivale à ativação da variável de estado de mesmo nome v, assim como a ausência da marcação equivale a $\neg v$.

Demonstrando que toda e qualquer ativação de transições possível corresponda a uma ação válida em P para os valores das variáveis de estado definidas por DTGs V_d: Cada transição $o = \langle P, E \rangle$ é também um operador em O. Pela definição de DTG, se existe um arco $\langle p_i, o \rangle$, P contém p_i, e se existe um arco $\langle o, p_f \rangle$, E contém p_f. Assim, para uma transição que remove o lugar p_i e adiciona o lugar p_f, existe um operador que desativa a variável de estado p_i e adiciona a variável de estado p_f.

Continuando a construção da rede, todas as redes derivadas de DTGs são combinadas em uma única rede. Transições com o mesmo nome são combinadas, mantendo todos os arcos de entrada e saída; A rede continua segura, pois o número de arcos de entrada de cada transição sempre é igual ao número de arcos de saída. Se duas redes derivadas

de DTGs diferentes possuírem lugares provindos da mesma variável, estes podem ser combinados com arcos repetidos entre o mesmo par de lugar e transição eliminados para manter a propriedade de uma marcação por lugar na rede.

Assim, este processo produz uma rede Condição/Evento a partir do problema de planejamento com um lugar para cada variável de estado representada por um DTG e uma transição para cada operador que afete uma destas variáveis de estado. Esta rede tem a metade do número de lugares da rede construída com o *Petrify* para o mesmo conjunto de variáveis de estado, uma vez que cria somente um lugar para cada variável de estado do problema ao invés de dois, e além disso, não duplica transições para trabalhar com os lugares de variáveis negadas que seriam adicionados.

Para as variáveis de estado não pertencentes a DTGs, é utilizado o algoritmo de conversão do *Petrify*. Procede-se à descrição do algoritmo que produz as variáveis multivaloradas e os DTGs e integra a rede construída a partir destes com a produzida pelo *Petrify* para as variáveis de estado booleanas.

4.2 Algoritmo

Havendo provado a possibilidade de criação de redes de Petri através de grafos de transição de domínio, procede a descrição completa do algoritmo de resolução destes problemas.

Inicialmente, seguindo o método descrito na seção 3.5, são obtidos os predicados constantes do domínio e o conjunto de variáveis de estado do problema. Procede-se à geração de varíaveis multivaloradas e, conforme descrito na seção 3.6, seus respectivos DTGs.

Variáveis de estado que não pertencem a DTGs são representadas pelo método usado pelo *Petrify* (seção 3.4), nos quais cada variável de estado é representada por um laço de dois lugares com marcação mutuamente exclusiva.

A seguir, garante-se que as redes com variáveis de estado que não estão em DTGs são seguras. Utiliza-se o método do *Petrify* (seção 3.4) para isso. Para cada variável de estado $a \in A$ que precisa ser representada desta maneira, é criada uma variável de estado $\hat{a}$ que é verdadeira sempre que a for falsa e vice-versa. Para garantir isto, todas as ocorrências de a em efeitos de operadores são substituídas pelo par $\{a, \neg\hat{a}\}$ e todas as ocorrências de $\neg a$ são substituídas pelo $\{\neg a, \hat{a}\}$.

Operadores que alteram o valor de uma variável de estado independentemente do seu valor anterior precisam ser duplicados caso esta variável de estado tenha sido duplicada, como demonstrado no artigo de Hickmott *et al.* Para cada operador $o = \langle p, e\rangle$ onde $(a \in e \vee \neg a \in e)$ e $(a \notin p \wedge \neg a \notin p)$, e a não está em nenhum DTG, o operador é substituído pelos operadores $o' = \langle p \cup a, e \backslash a\rangle$ e $o'' = \langle p \cup \neg a, e\rangle$. Um único operador pode ser duplicado várias vezes se mais de uma das variáveis de estado afetadas tiver sido duplicada.

Finalmente, uma nova busca é feita na rede de Petri para localizar lugares e transições inalcançáveis a partir da marcação inicial. Este último passo elimina lugares provindos de predicados de caminho único que não são utilizados pelo problema, bem como quaisquer outras seções da rede que não sejam relacionadas à solução. A busca é feita da mesma forma que aquela descrita na seção 3.5:

1. Define-se o grupo G como o grupo de lugares marcados inicialmente na rede de Petri;

2. Para cada lugar ou transição i em G, adiciona-se a G todos os lugares e transições nos quais chega um arco saindo de i. Esse passo é repetido até não existirem lugares ou transições a serem adicionados;

3. Todos os lugares e transições da rede de Petri não pertencentes a G podem ser eliminados da rede.

O resultado é uma rede Condição-Evento na qual o método de desdobramento pode ser aplicado, da mesma forma que o *Petrify*, mas de tamanho consideravelmente menor, como mostrado a seguir.

4.2.1 Comparação com *Petrify*

Cada um dos passos nos quais o método proposto difere do *Petrify* mantém ou reduz o tamanho da rede de Petri gerada. Sendo A o conjunto de variáveis de estado do problema, a rede gerada pelo *Petrify* tem $2 \cdot (|A|)$ lugares (Seção 3.4).

Já a rede gerada pelo *Petrigraph* tem tantos lugares quanto a criada pelo Petrify, menos os lugares de variáveis de estado constantes, lugares variáveis de estado negadas se estas variáveis estão em DTGs, e os lugares inacessíveis. Assim, a rede gerada pelo *Petrigraph* tem $2 \cdot (|A - A_s|) - |A_D| - p_u$ lugares, onde $A_s \in A$ é o conjunto de variáveis de estado constantes, $A_D \in A - A_s$ é o conjunto de variáveis de estado que pertencem a DTGs e $p_u \geq 0$ é o número de lugares da rede de Petri inacessíveis da configuração inicial.

Passando à análise de número de transições, a rede gerada pelo *Petrify* tem $\sum^i 2^{v(O_i)}$ transições, onde $v(O_i)$ é o número de variáveis de estado que são afetadas pelo operador O_i mas não estão incluídos na lista de pré-condições deste (Seção 3.4).

Quando uma variável de estado está em um DTG, os operadores que removem aquela variável sem verificar o seu valor não são duplicados para cobrir a possibilidade desta variável estar negada. Assim, a rede gerada pelo *Petrigraph* é composta de $\sum^i 2^{v(O_i)-v_D(O_i)}$

transições, onde $v_D(O_i)$ é o número de variáveis de estado em DTGs que são afetadas pelo operador O_i mas não estão incluídos na lista de pré-condições deste.

Assim, é demonstrado que a rede de Petri gerada por este método pode apenas ser menor ou igual em tamanho, tanto em lugares quanto em transições, à rede gerada a partir do mesmo problema pelo *Petrify*.

4.2.2 Desdobramento no *Petrigraph*

Uma implementação de desdobramento como o `MOLE` produz o prefixo finito completo da rede de Petri gerada pelo algoritmo. Uma vez que não precisamos do desdobramento completo, mas apenas um caminho para o objetivo, o algoritmo pode ser otimizado para uma busca orientada a objetivo, ao invés de uma busca exaustiva. Para tanto, será explorada a heurística h^1, descrita na seção 3.7.

Para terminar a busca uma vez que a condição de objetivo é alcançada, toma-se vantagem da capacidade do `MOLE` de parar quando alcança uma determinada transição. É criada uma transição de objetivo cujo disparo depende de marcas em todos os lugares de objetivo da rede, e o algoritmo de desdobramento é instruído para que pare quando alcançar esta transição.

Normalmente, o `MOLE` ordena os itens a serem expandidos de tal maneira que cada camada é explorada inteiramente antes da próxima, fazendo o equivalente a uma busca em largura. Buscando otimizar o processo para encontrar o objetivo, a fila de expansão do `MOLE` foi reestruturada para usar a heurística monotônica h^1 [24, 25], recomendada como a mais eficiente no artigo de Hickmott *et al.* Cada lugar da rede de Petri recebe um peso de acordo com a sua distância mínima ao objetivo, e a fila de expansão é ordenada de acordo com a soma dos pesos dos lugares marcados em cada expansão.

4.3 Implementação

O *Petrigraph* foi implementado em linguagem C++. Inicialmente ele foi desenvolvido a partir do analisador gramatical (*parser*) PDDL e estruturas de dados da plataforma

Ipê [26], criada pelo Laboratório de Inteligência Computacional da UFPR com o intuito de padronizar uma biblioteca para implementação de algoritmos de planejamento. No entanto, a versão utilizada do analisador gramatical revelou ter dificuldades em ler boa parte dos domínios de teste. Assim, o algoritmo foi reconstruído a partir do analisador gramatical PDDL e estruturas de dados da implementação do Fast Forward por Hoffmann *et al* [27].

A busca por variáveis de estado do problema foi substituída por uma enumeração (*hashing*) das variáveis de estado com a intenção de simplificar a implementação e aumentar a velocidade. Assume-se que um predicado *pred* com argumentos $a_1 \ldots a_n$ define variáveis de estado para todos os valores possíveis de $a_1 \ldots a_n$. Nos domínios de PDDL com tipos testados, esta enumeração cria variáveis de estado inacessíveis a partir da situação de início; por exemplo, a enumeração do domínio ROVERS, onde apenas sondas equipadas para análise de solo podem adquirir dados de solo, a enumeração irá alocar uma posição para a variável de estado `tem_análise_do_solo (sonda_a, terreno_a)` mesmo que a variável de estado `capaz_de_análise_do_solo (sonda_a)` não esteja ativa. A remoção de predicados constantes e o corte de lugares inacessíveis no final do algoritmo garante que a rede criada não contém lugares não utilizados que sejam definidos desta forma.

De particular nota na implementação do algoritmo de combinação de predicados são os predicados sem argumentos, que precisaram ser tratados como casos especiais tanto na busca de predicados a serem combinados quanto na determinação de lugares a partir de DTGs.

Foi incluído o suporte a tipos do PDDL. Para estender o balanceamento a predicados com argumentos de tipos diferentes, foi definida a seguinte regra de união de predicados:

Definição 26 (União de predicados com tipagem). *Dois predicados* $pred(p_1, \ldots, p_n)$ *e* $pred'(p'_1, \ldots, p'_n)$ *só podem ser unidos se, para todo* m *onde* $1 < m < n$*, dados os tipos* T *e* T' *onde* $p_n \in T$ *e* $p'_n \in T's$*, é verdadeiro que* $T \subset T'$ *ou* $T' \subset T$*.*

O processo de criação de lugares e transições exigiu atenção especial. O processo é recursivo, ramificando-se para cada valor possível em cada argumento dos predicados e ações: No caso de operadores duplicados, a duplicação é feita durante este passo se

a necessidade foi determinada anteriormente. O uso de variáveis de estado constantes implica no tratamento especial de transições, uma vez que elas não poderão depender de lugares derivados destas variáveis. Uma busca na árvore de pré-condições gerada pelo parser PDDL define se a transição depende da variável de estado constante estar ativada ou desativada, e compara com o valor inicial da variável no problema (que, por ser constante, não irá mudar durante a resolução). Se a transição for inacessível, ela não é adicionada à rede. A mesma busca na árvore de pré-condições determina os arcos que irão ligar a transição criada aos lugares da qual ela depende.

A implementação do *Petrigraph* foi feita de forma que algumas etapas do processo sejam opcionais, o que permite inferir a influência de cada etapa no desempenho final do sistema. As etapas opcionais são a criação de variáveis multivaloradas e DTGs, a busca e remoção de predicados constantes, e a busca e corte[1] de lugares inacessíveis. Com estas três etapas removidas, o algoritmo é funcionalmente equivalente ao *Petrify* conforme descrito no artigo de Hickmott *et al.*

O programa produz arquivos no formato `LL_NET` usado pelo `MOLE`. Inicialmente o cálculo da heurística h^1 era feito no módulo de *Petrigraph*, o que exigiu uma alteração no formato `LL_NET` e no `MOLE`. Isso tornou os arquivos no formato `LL_NET` alterado incompatíveis com o `MOLE` normal, e vice-versa. Assim, o cálculo da heurística foi passado para o `MOLE` e o arquivo `LL_NET` foi retornado ao seu formato de base. Além disso, o `MOLE` foi alterado para ter a capacidade de terminar automaticamente a execução após um certo tempo ou número de expansões.

O capítulo a seguir irá detalhar os testes executados e seus resultados.

[1] *Cull.*

CAPÍTULO 5

TESTES

5.1 Algoritmos

Foi testada a implementação do *Petrigraph* com todas as combinações possíveis de três passos opcionais: a criação de variáveis multivaloradas e DTGs, a busca e remoção de predicados constantes, e a busca e remoção de lugares inacessíveis (*cull*). Com estas três etapas removidas, o algoritmo é funcionalmente equivalente ao *Petrify* conforme descrito no artigo de Hickmott *et al.* Assim, esse algoritmo foi usado como a implementação do *Petrify* para os testes efetuados, uma vez que o código original do *Petrify* utilizado no artigo de Hickmott *et al* não estava disponível.

Testes foram feitos com o *MOLE* utilizando tanto uma busca por largura quanto a heurística h^1. Esta última é a recomendada na combinação entre *Petrify* e *MOLE*, formando o algoritmo denominado *PUP*. Como comparação, é utilizada a implementação *SatPlan2006* do algoritmo *SatPlan* por Kautz e Selman[1]

Para o algoritmo *SatPlan* foi medido o tempo de execução. Para os algoritmos *Petrify* e *Petrigraph* foram medidos o tempo de execução e o número de expansões feitas pelo *MOLE*. O tempo de execução dos algoritmos foi limitado a 10 minutos, exceto quando indicado no texto. Pontos que não aparecem nos gráficos não o fizeram por quebra do limite de tamanho de rede criada pelo *Petrify* ou *Petrigraph*, ou pela quebra do limite de tempo de execução.

5.2 Domínios

Os domínios de planejamento descritos aqui foram utilizados nas competições de planejamento AIPS-2000 [28] e IPC-2002 [29]. Os domínios são: BlocksWorld, Elevator,

[1]Disponível em `http://www.cs.rochester.edu/ kautz/satplan/` - Último acesso: 19 Mar. 2008.

LOGISTICS, DRIVERLOG, ZENOTRAVEL, SATELLITE e ROVERS. Estes domínios foram utilizados para a comparação de desempenho de algoritmos de planejamento heterogêneos, e são considerados pela comunidade científica como bons representantes de problemas de planejamento em geral. Os domínios aparecem aqui com seus nomes em inglês, como nomeados nas competições.

BLOCKSWORLD é um dos problemas clássicos de planejamento. Ele captura várias das dificuldades inerentes a sistemas de planejamento em um domínio relativamente simples. O domínio BLOCKSWORLD consiste em um número arbitrário de blocos similares empilhados sobre uma superfície (ou mesa) de tamanho indefinido. Cada bloco pode ter no máximo um bloco diretamente apoiado nele, e pode se apoiar em cima de um único bloco se não estiver diretamente sobre a mesa. A operação do sistema se dá por uma garra, capaz de capturar um bloco que não esteja obstruído e depositá-lo em cima de outro bloco ou na mesa. A complexidade do BLOCKSWORLD deve-se principalmente ao grande número de *deadlocks* e a necessidade de se alcançar os objetivos em uma ordem correta para se ter a estratégia ótima [30].

O domínio ELEVATOR, ou MICONIC-10 [31] replica o sistema de elevadores de um edifício. Por ser um recurso relativamente limitado em relação ao número de usuários e dispendioso de energia, a otimização de uso de elevadores é fundamental. O domínio simula a operação de um único elevador em um edifício com um número de andares e usuários arbitrário. A cada passageiro são associados predicados indicando o seu andar de origem e destino, bem como se ele está no elevador e se já está em seu destino. As ações do sistema consistem na entrada e saída de passageiros e na movimentação do elevador para cima e para baixo, um andar por vez.

O domínio LOGISTICS [32] simula o problema de transportar cargas entre localidades. O problema de transporte de pessoal por elevador pode ser generalizado para o transporte de qualquer carga, para situações de qualquer escala onde se exija a movimentação de carga de um ponto a outro com o menor dispêndio de recursos possível. Localidades estão situadas em cidades; para o transporte das cargas, estão à disposição caminhões que podem ser movidos entre localidades da mesma cidade, bem como aviões que podem

mover-se para qualquer aeroporto (um tipo especial de localidade) do sistema, independente de qual cidade ele se localiza.

O domínio DRIVERLOG é similar ao LOGISTICS, com caminhões transportando cargas entre localidades, mas com a complicação que os caminhões requerem motoristas para dirigir entre estas localidades. Os caminhos para motoristas e caminhões formam mapas diferentes entre as localidades.

O domínio ZENOTRAVEL define aviões que podem transportar passageiros entre localidades, com velocidades definidas independentemente para cada avião. A qualidade do domínio ZENOTRAVEL é dependente do uso de pesos numéricos. No modo Strips, o modo que é utilizado para estes testes, ele restringe-se a um domínio adicional de transporte.

SATELLITE foi inspirado por aplicações espaciais e é um primeiro passo para a *espaçonave ambiciosa* descrita por David Smith *et al* [33]. O domínio envolve distribuir uma coleção de trabalhos de observação entre satélites diferentes, cada um equipado de maneira diferente.

ROVERS requer que uma coleção de veículos exploradores autônomos (*rovers*) naveguem a superfície de um planeta, encontrando amostras e trazendo-as de volta para um módulo de pouso. A versão Strips deste domínio tenta implementar um dispositivo anti-concorrência fazendo com que certas ações removam uma variável de estado e imediatamente a adicionem de volta. Esse dispositivo não se adapta à conversão para rede de Petri, e foi removido durante os testes.

A intenção inicial era testar os domínios FREECELL e SCHEDULE, mas estes domínios geram grandes quantidades de variáveis de estado, e assim, redes de Petri extremamente grandes relativamente ao tamanho do problema quando comparadas a outros domínios, tornando impossível executar uma bateria de testes exaustivos em tempo hábil.

DTG	corte	constante	BlocksWorld		Elevator		DriverLog		ZenoTravel	
			P	T	P	T	P	T	P	T
			262	1241	1680	2161	1284	5101	190	41625
X			262	221	1680	1401	1284	2521	190	13001
	X		262	1241	880	791	753	1105	141	1457
X	X		140	221	860	411	564	517	109	465
		X	262	1241	80	791	384	2077	92	1457
X		X	262	221	80	411	384	949	92	465
	X	X	262	1241	80	791	303	1105	92	1457
X	X	X	140	221	60	411	114	517	60	465

Tabela 5.1: Comparação de lugares e transições na rede de Petri gerada

5.3 Resultados

Os gráficos de resultados estão arranjados por tamanho do problema, e quando este valor não estava disponível, pelo número identificador na competição a que pertencem. Problemas múltiplos com o mesmo tamanho estão arranjados em posições fracionárias no eixo de identificação, de forma que estejam adjacentes no gráfico.

5.3.1 Tamanho da rede

A tabela 5.1 compara o número de lugares e transições na rede de Petri gerada para um problema de tamanho médio (dentre os fornecidos na competição) para quatro domínios: BlocksWorld, Elevator, DriverLog e ZenoTravel. As marcações nas linhas horizontais mostram quais passos do *Petrigraph* foram ativados; O algoritmo usado para os dados na primeira linha é fundamentalmente idêntico ao *Petrify*, enquanto o algoritmo para os dados na última linha é o *Petrigraph* completo.

Pode-se ver que todos os três passos têm efeitos no tamanho da rede. Ainda, é possível ver que no domínio ZenoTravel, o *Petrigraph* gera uma rede com aproximadamente 3 vezes menos lugares, e aproximadamente 100 vezes menos transições.

Os gráficos a seguir mostram tamanhos de rede para todos os problemas analisados.

Em todos os casos, o *Petrigraph* consistentemente cria redes de tamanho menor que o *Petrify*, tanto em número de lugares quanto de transições. Os passos do processo que reduzem a rede dependem do domínio em particular: o domínio BlocksWorld não

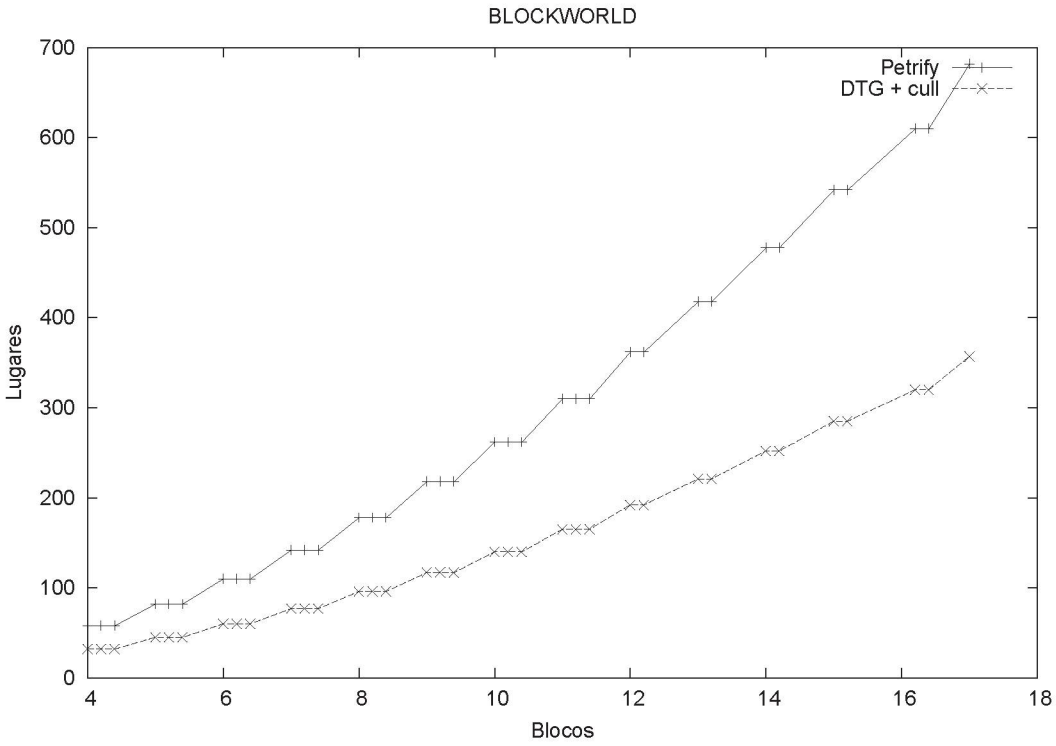


Figura 5.1: Número de lugares para o domínio BLOCKSWORLD

tem predicados constantes a serem eliminados, e beneficia-se principalmente da criação de DTGs. O domínio ELEVATOR tem grande número de predicados constantes e poucos convertíveis para DTGs, e o corte de constantes é mais importante para a redução do tamanho da rede. Já o domínio LOGISTICS tem tanto predicados constantes quanto predicados utilizáveis em DTGs, e beneficia-se de ambos.

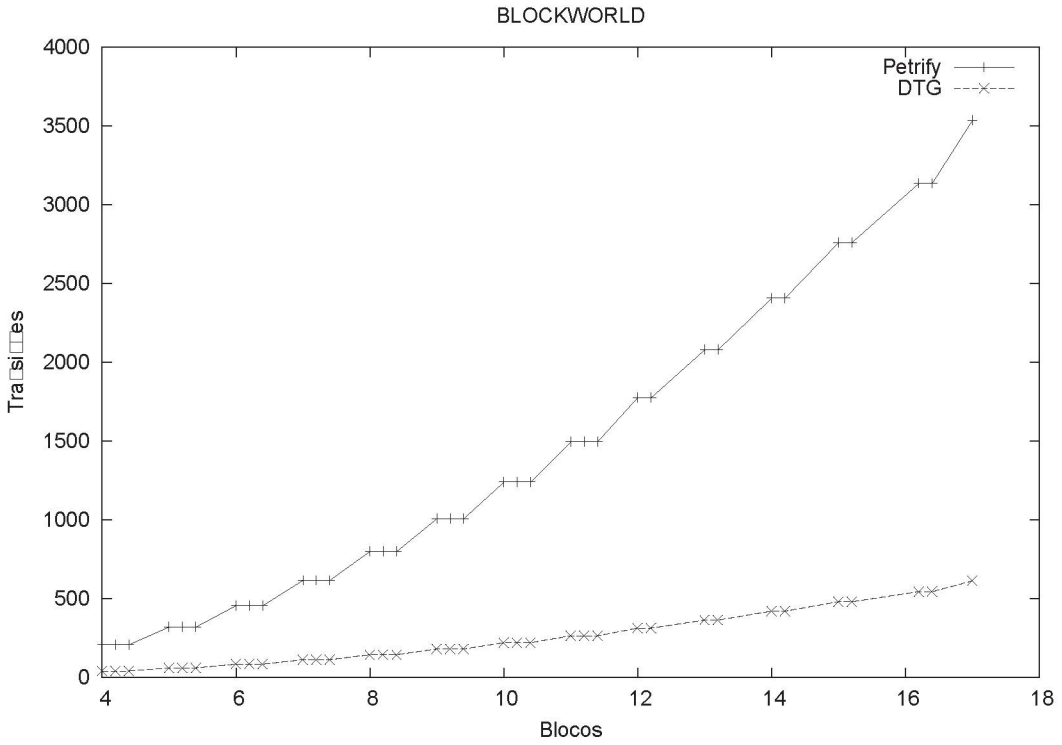


Figura 5.2: Número de transições para o domínio BlocksWorld

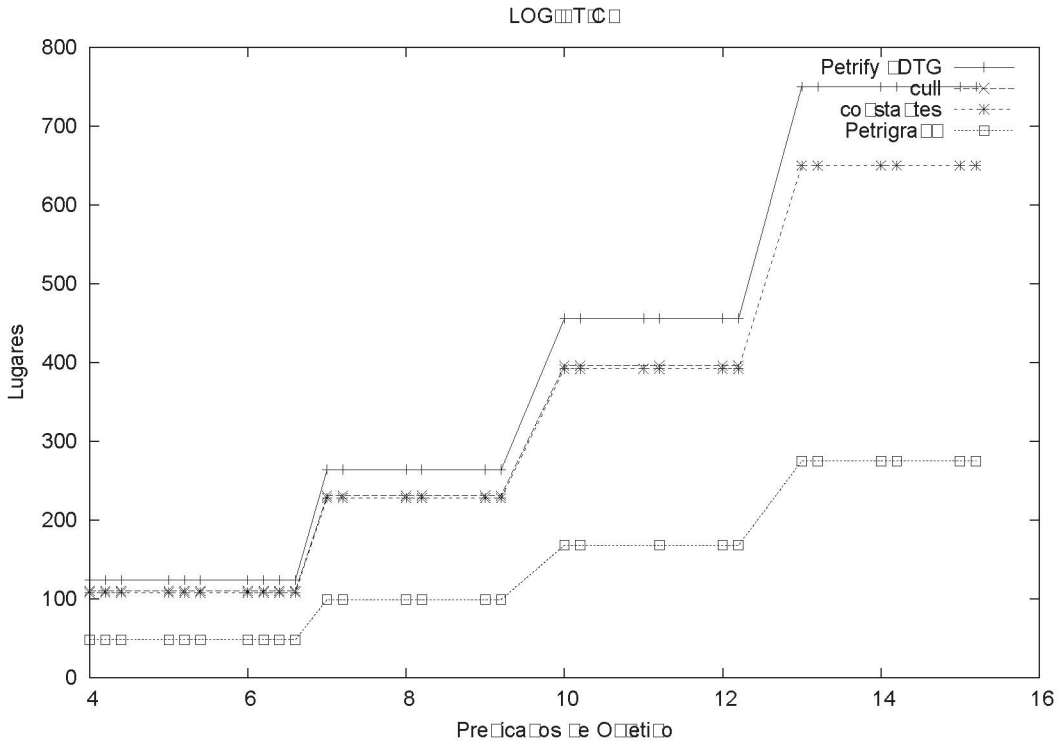


Figura 5.3: Número de lugares para o domínio Logistics

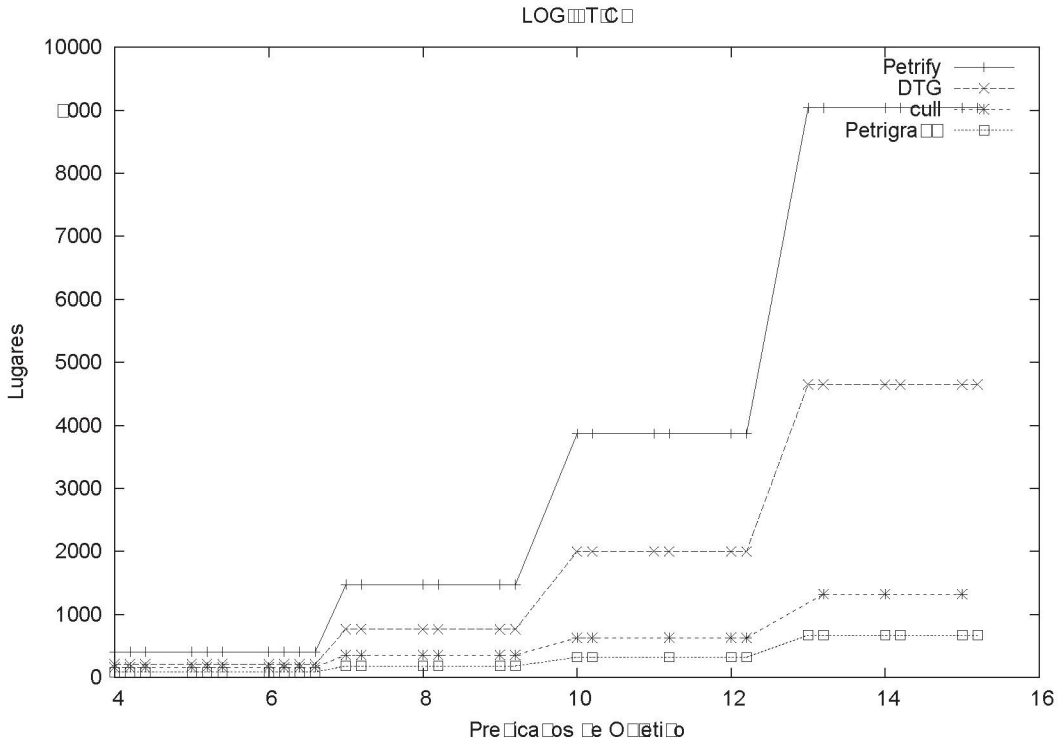


Figura 5.4: Número de transições para o domínio LOGISTICS

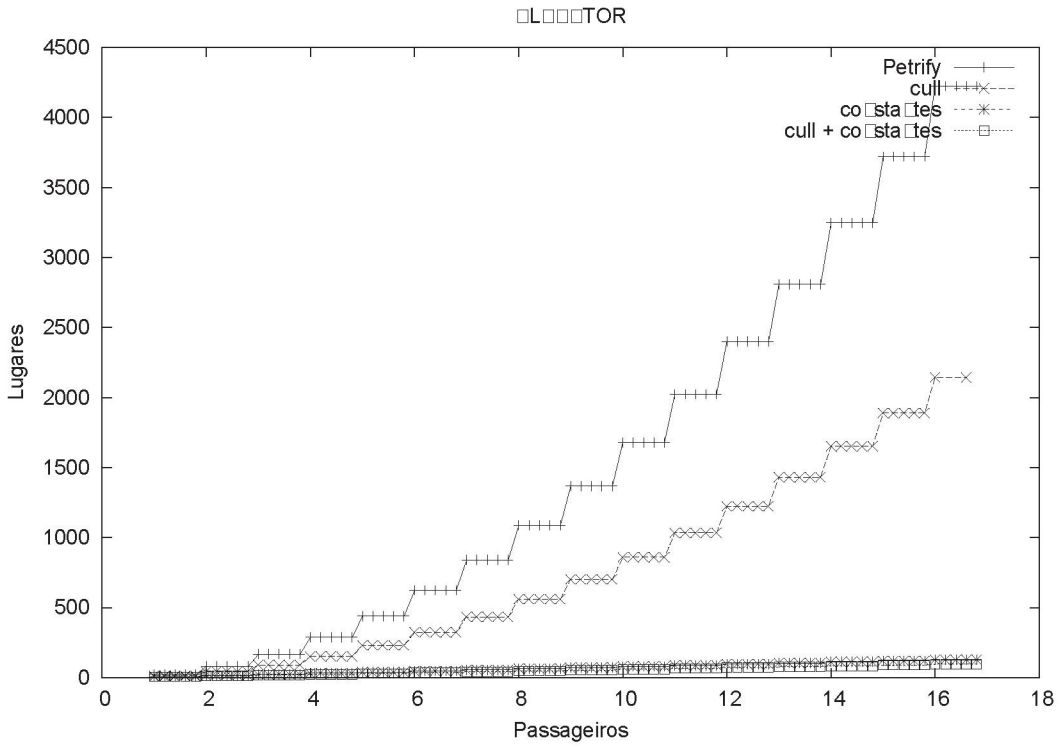


Figura 5.5: Número de lugares para o domínio ELEVATOR

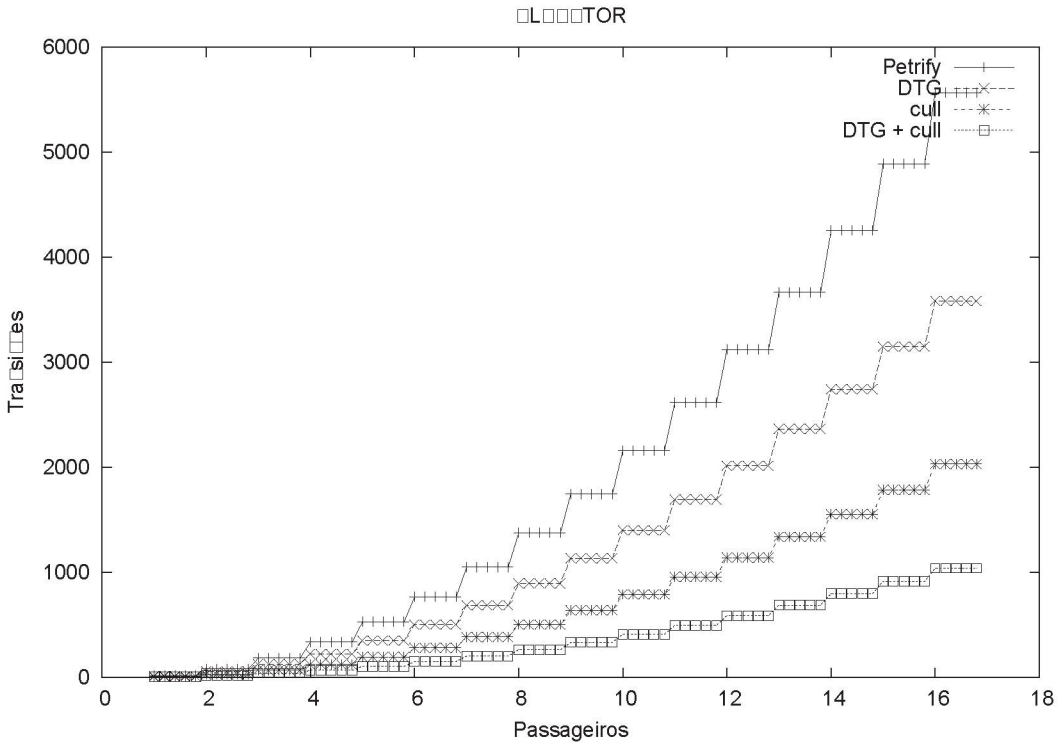


Figura 5.6: Número de transições para o domínio ELEVATOR

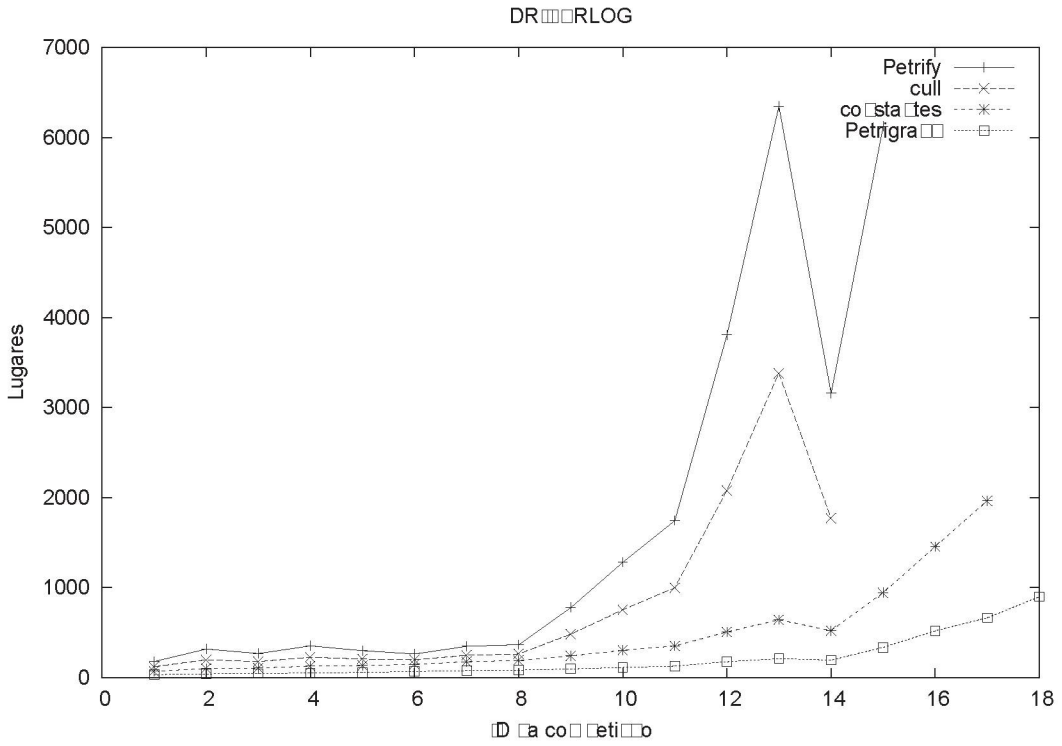


Figura 5.7: Número de lugares para o domínio DRIVERLOG

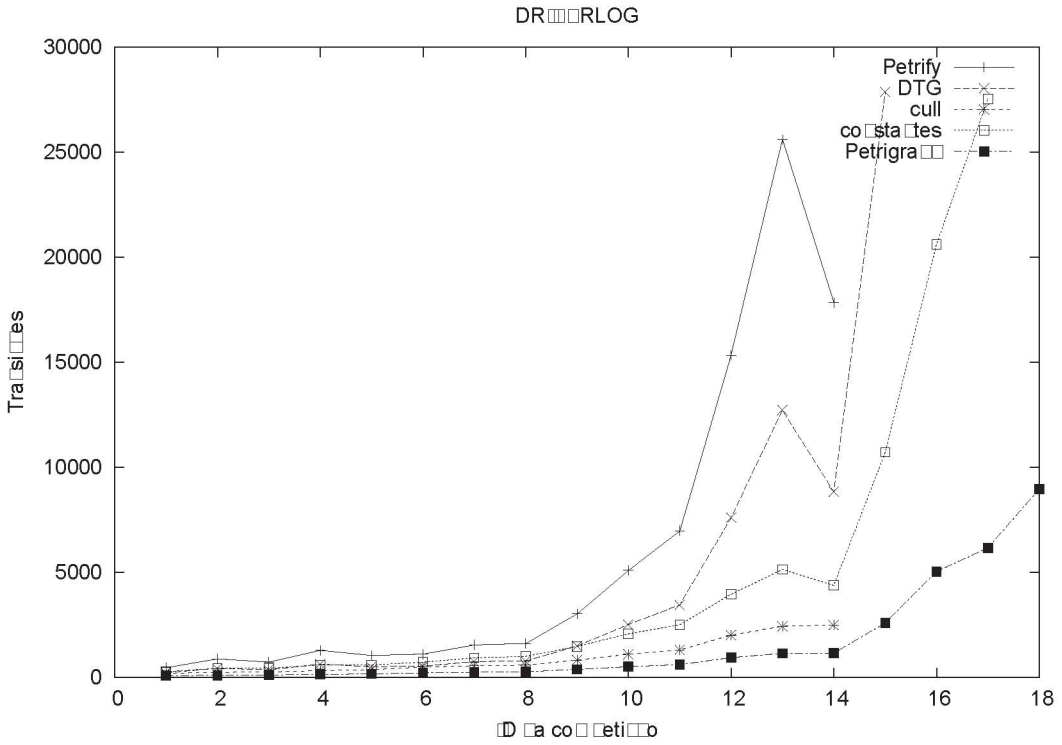


Figura 5.8: Número de transições para o domínio DriverLog

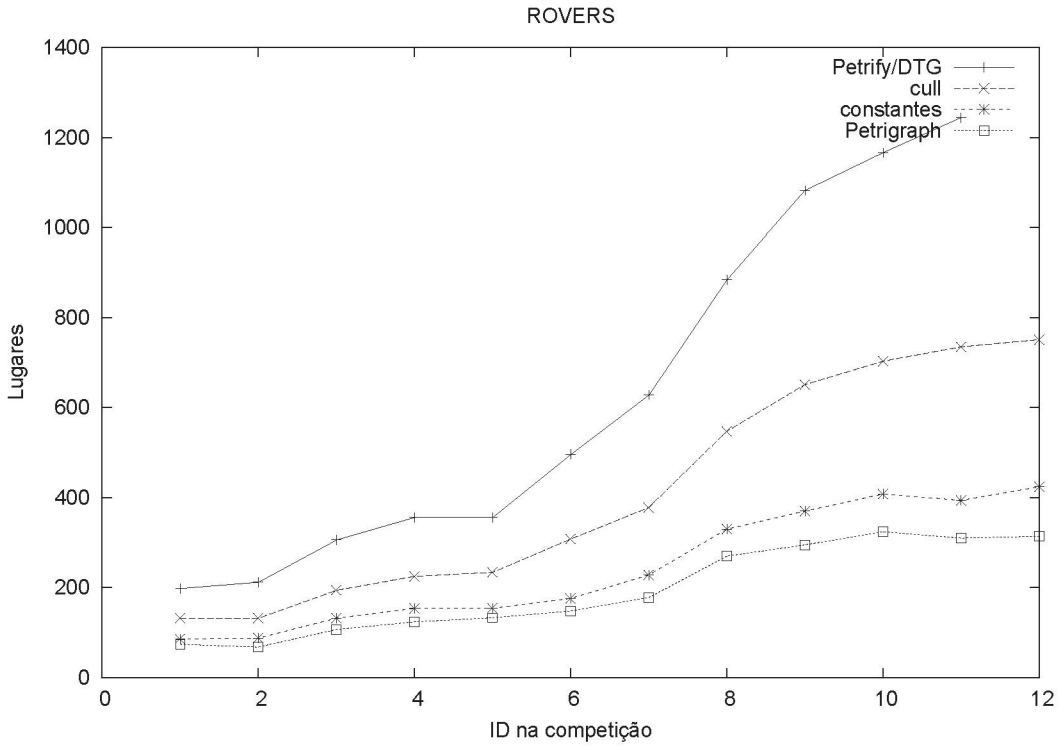


Figura 5.9: Número de lugares para o domínio Rovers

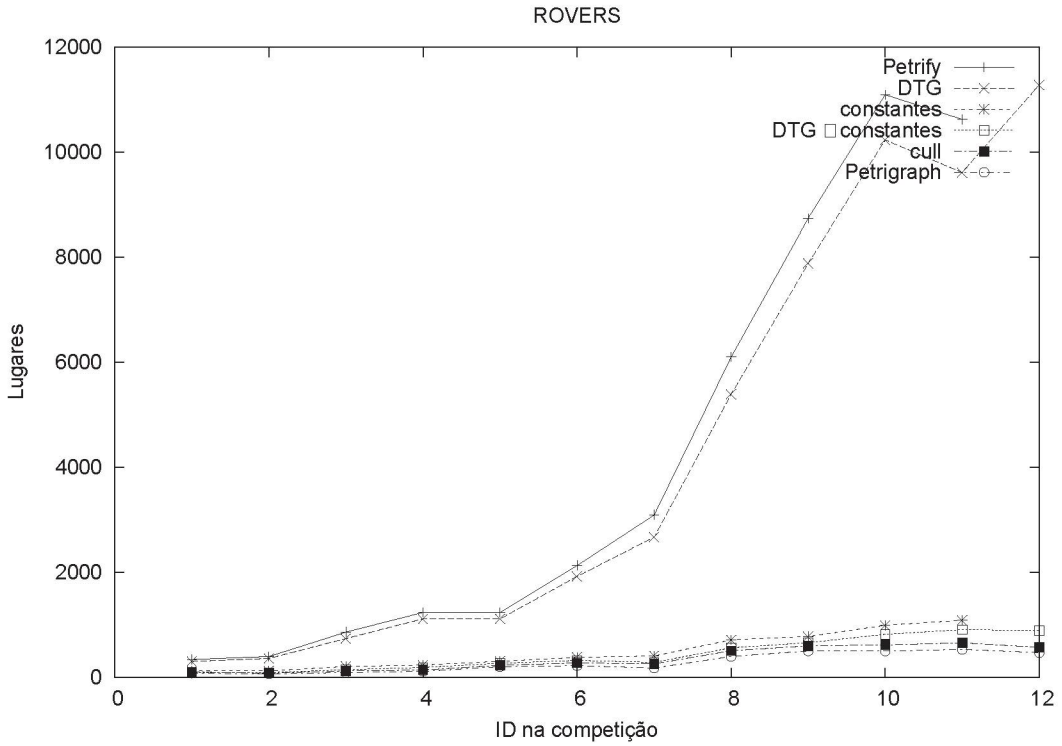


Figura 5.10: Número de transições para o domínio ROVERS

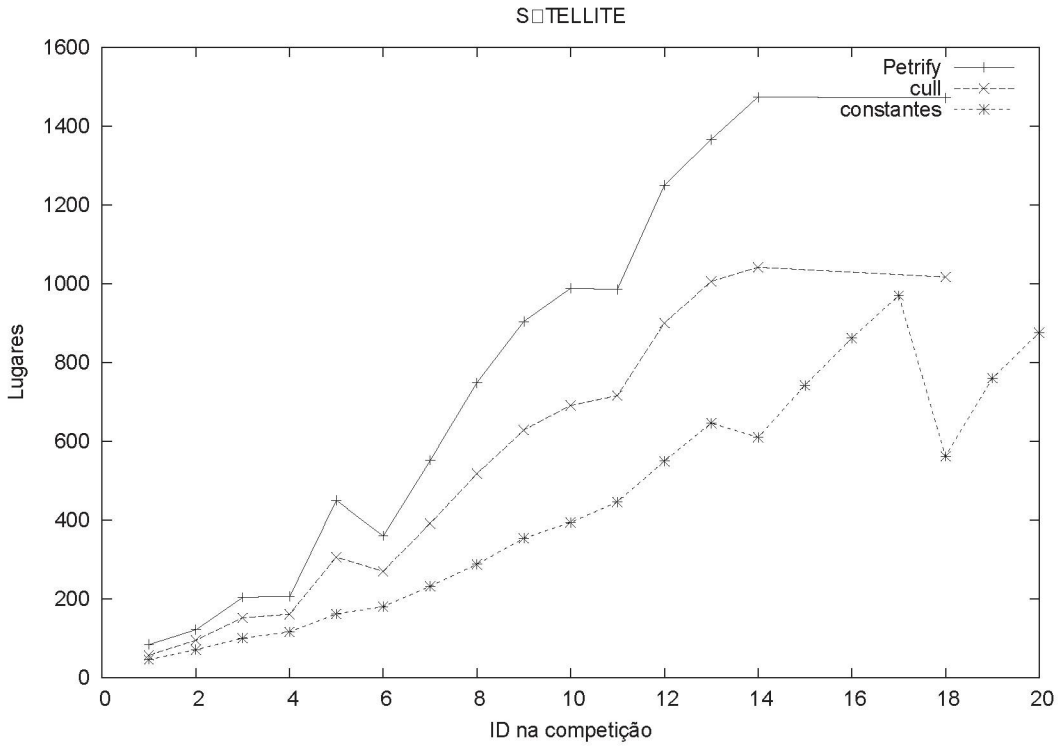


Figura 5.11: Número de lugares para o domínio SATELLITE

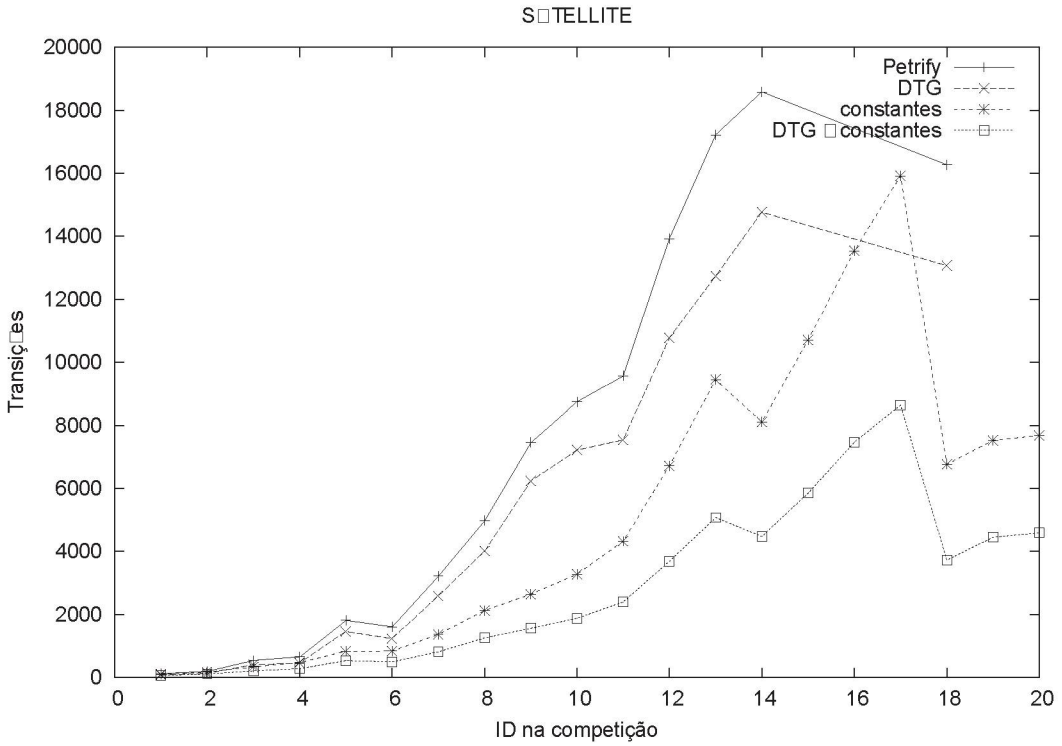


Figura 5.12: Número de transições para o domínio SATELLITE

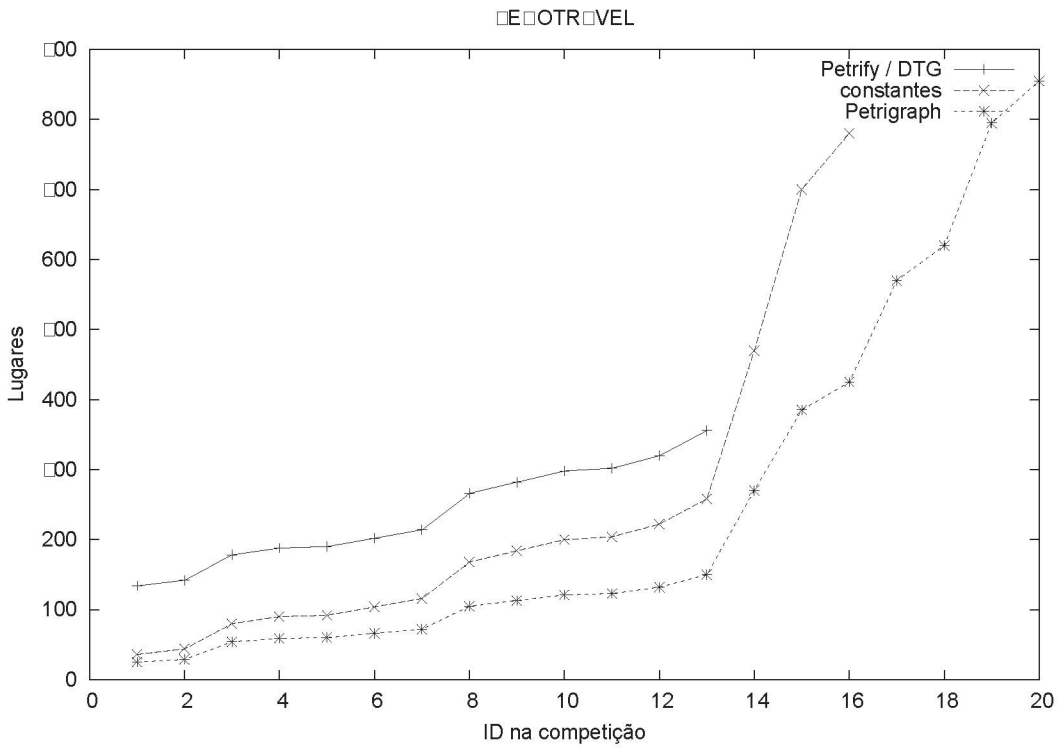


Figura 5.13: Número de lugares para o domínio ZENOTRAVEL

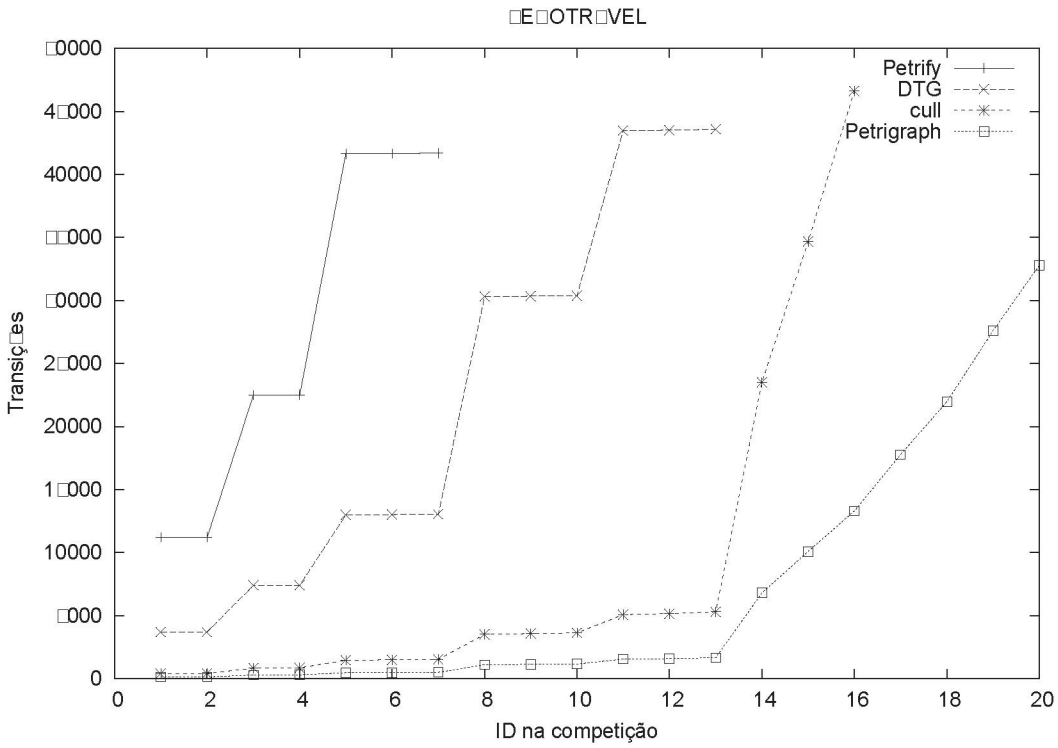


Figura 5.14: Número de transições para o domínio ZENOTRAVEL

5.3.2 Expansões e tempo de execução, sem heurística

Nesta bateria de gráficos, são mostrados os resultados apenas para o *Petrify*, *Petrify* com uso de DTGs, e o *Petrigraph* completo.

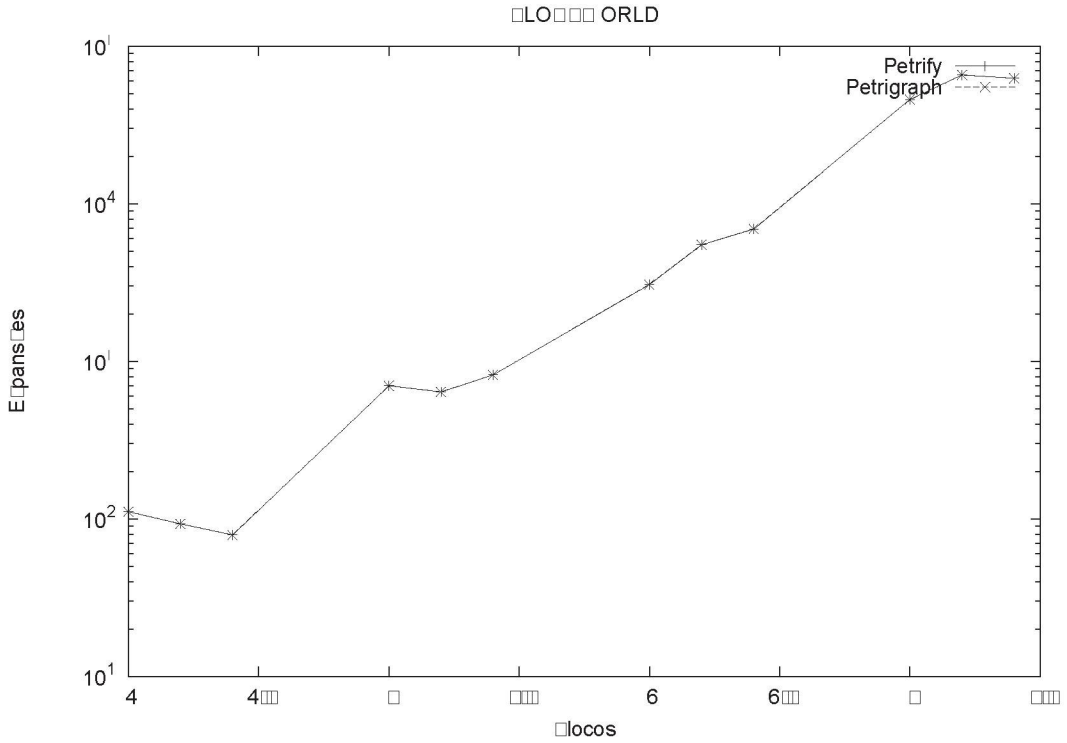


Figura 5.15: Expansões para o domínio BLOCKSWORLD, sem heurística

Nos domínios BLOCKSWORLD, LOGISTICS e ELEVATOR, o número de expansões é sempre o mesmo tanto em redes geradas pelo *Petrify* quanto em redes geradas pelo *Petrigraph*, o que leva-nos a concluir que a redução de tamanho da rede não altera o processo de busca. Com o número de expansões constante, o tempo de execução é estritamente dependente do tamanho da rede, que diminue conforme visto na seção 5.3.1. Nos três casos, o tempo de execução para o *Petrigraph* é mais de 10 vezes melhor que o para o *Petrify*.

A maioria dos problemas do domínio SATELLITE não são solucionados no tempo determinado sem heurística. Os três problemas que são solucionados demonstram expansões constantes e redução de tempo no *Petrigraph*, embora em menor escala que os domínios anteriores.

O domínio DRIVERLOG tem a peculiaridade de exigir menos expansões quando re-

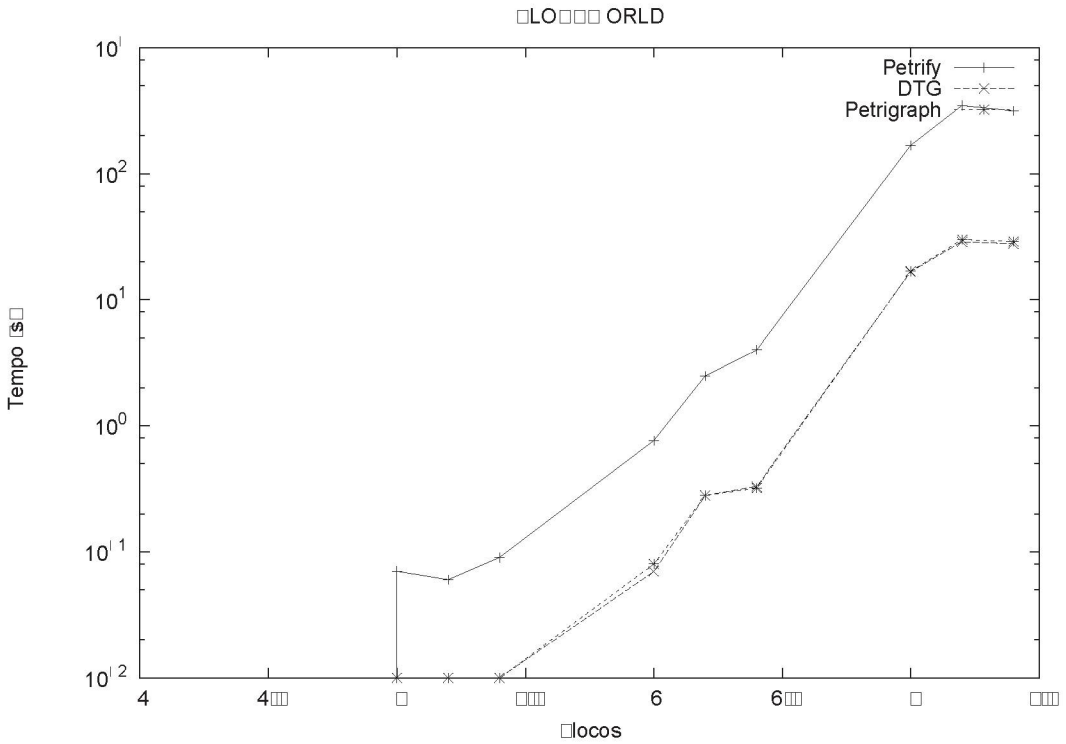


Figura 5.16: Tempo de execução para o domínio BLOCKSWORLD, sem heurística

solvido a partir da rede gerada pelo *Petrigraph*. É conjecturado que isso ocorre pelas características do método que o *MOLE* utiliza para armazenar a rede desdobrada em memória, já que este fenômeno não ocorreu nos domínios que não permitem ações concorrentes ou no domínio LOGISTICS.

O domínio ROVERS tem a mesma peculiaridade do domínio DRIVERLOG. O uso de DTGs torna a execução ligeiramente mais lenta nos primeiros casos, mas o *Petrigraph* completo continua sendo mais eficiente em termos de tempo.

O domínio ZENOTRAVEL passa pelo mesmo fenômeno dos domínio ROVERS e o domínio DRIVERLOG. Aqui, a vantagem do uso do *Petrigraph* é ainda mais dramática, chegando a um aumento de velocidade de mais de 100 vezes comparado ao *Petrify*.

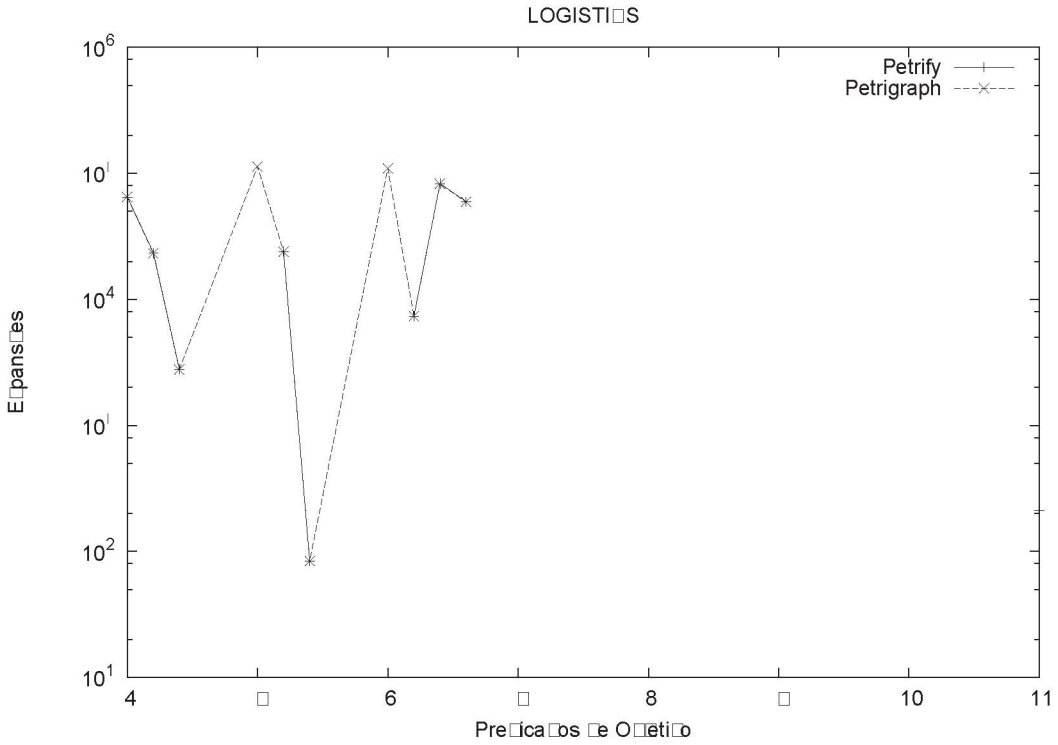


Figura 5.17: Expansões para o domínio LOGISTICS

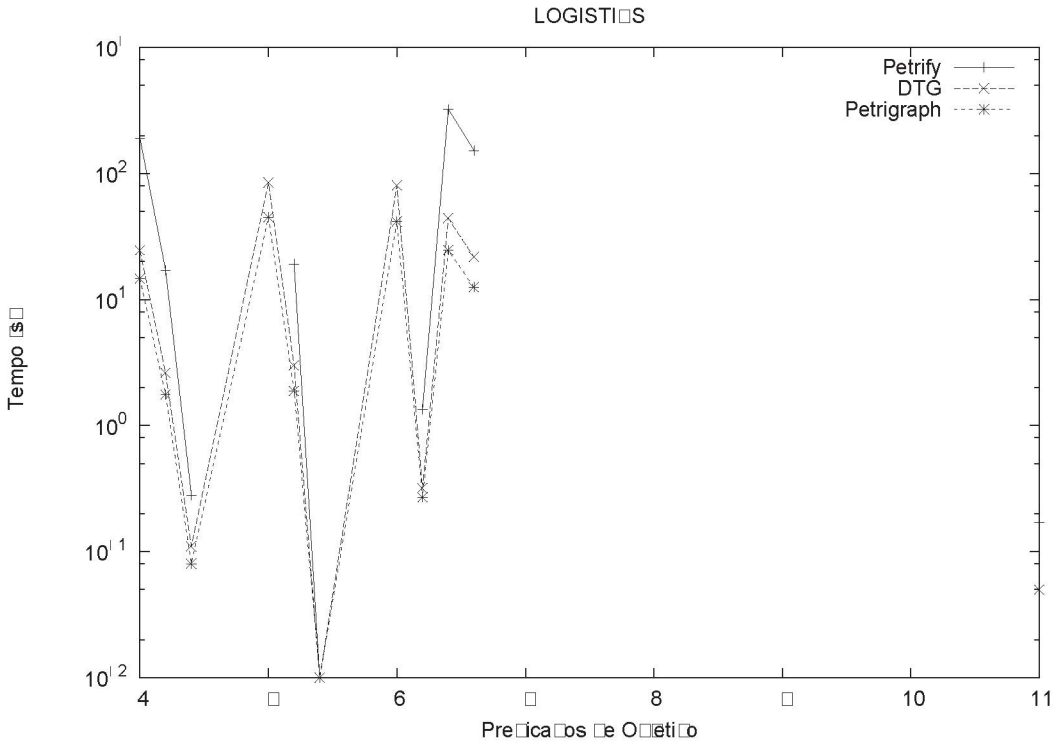


Figura 5.18: Tempo de execução para o domínio LOGISTICS

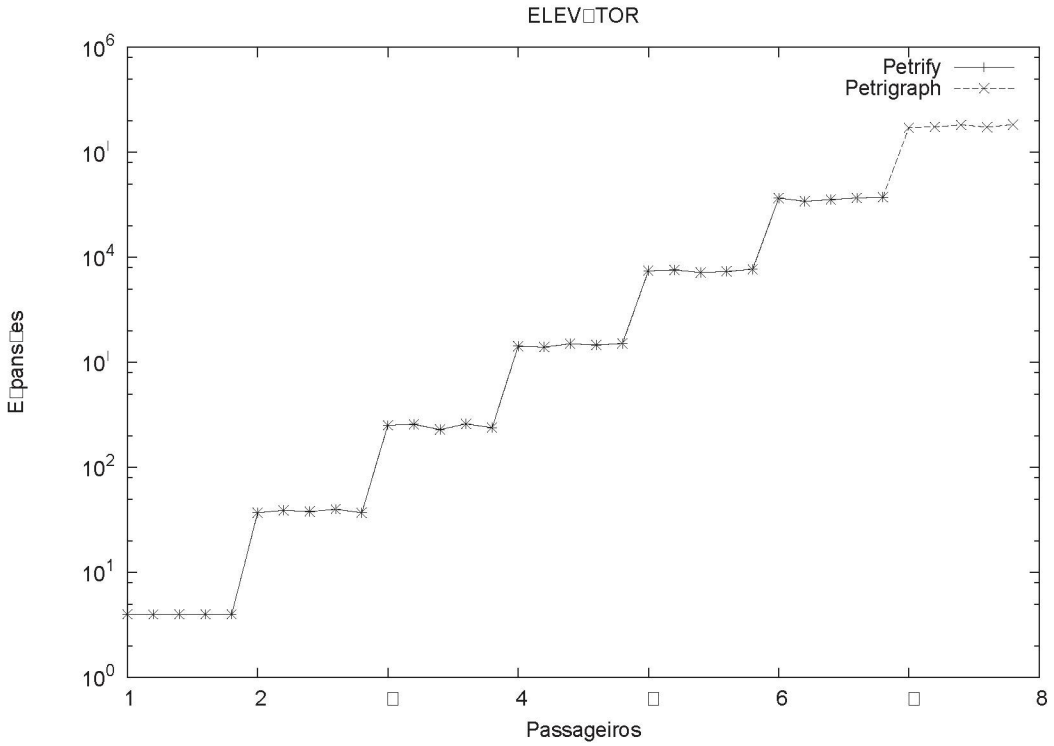


Figura 5.19: Expansões para o domínio ELEVATOR

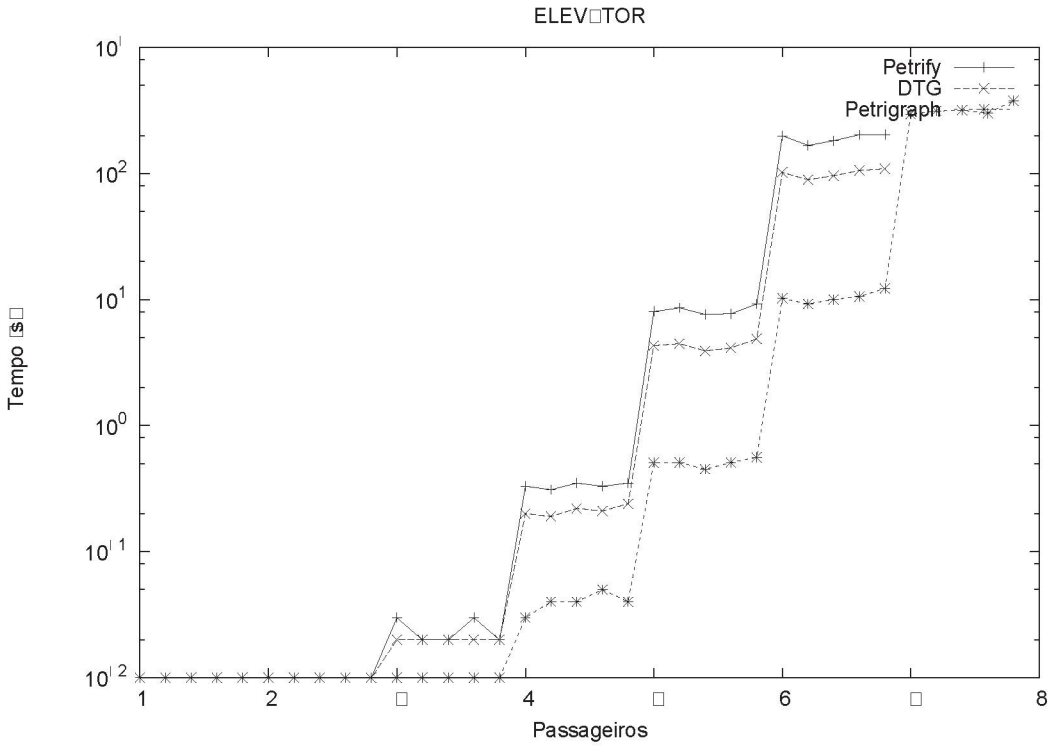


Figura 5.20: Tempo de execução para o domínio ELEVATOR

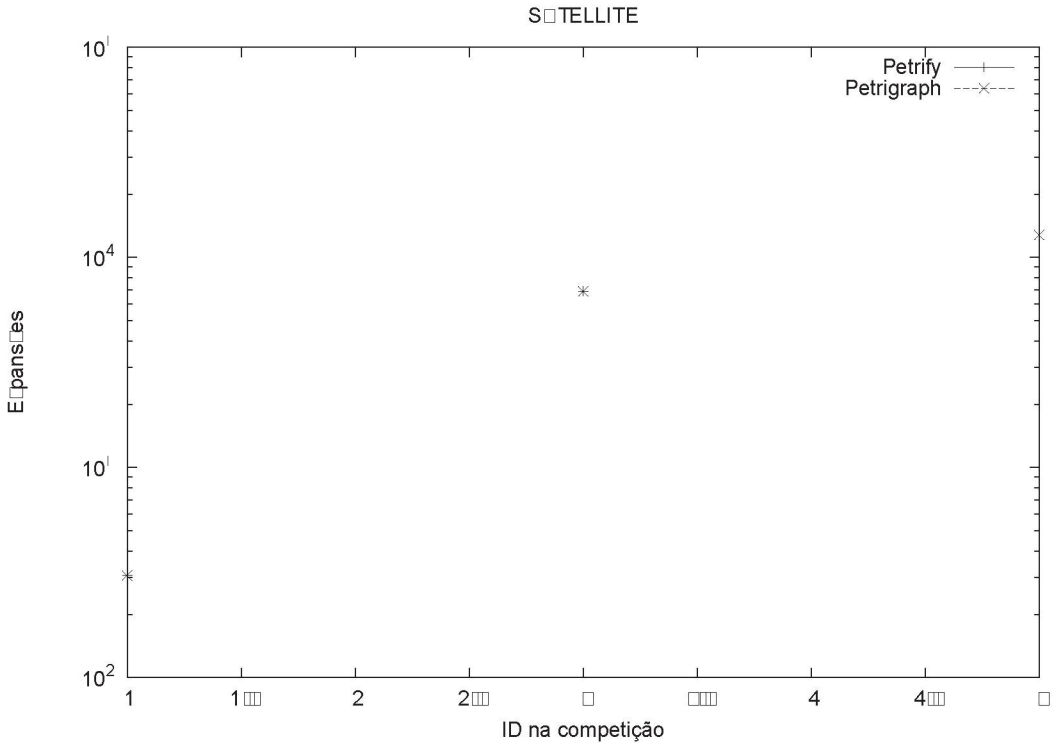


Figura 5.21: Expansões para o domínio SATELLITE

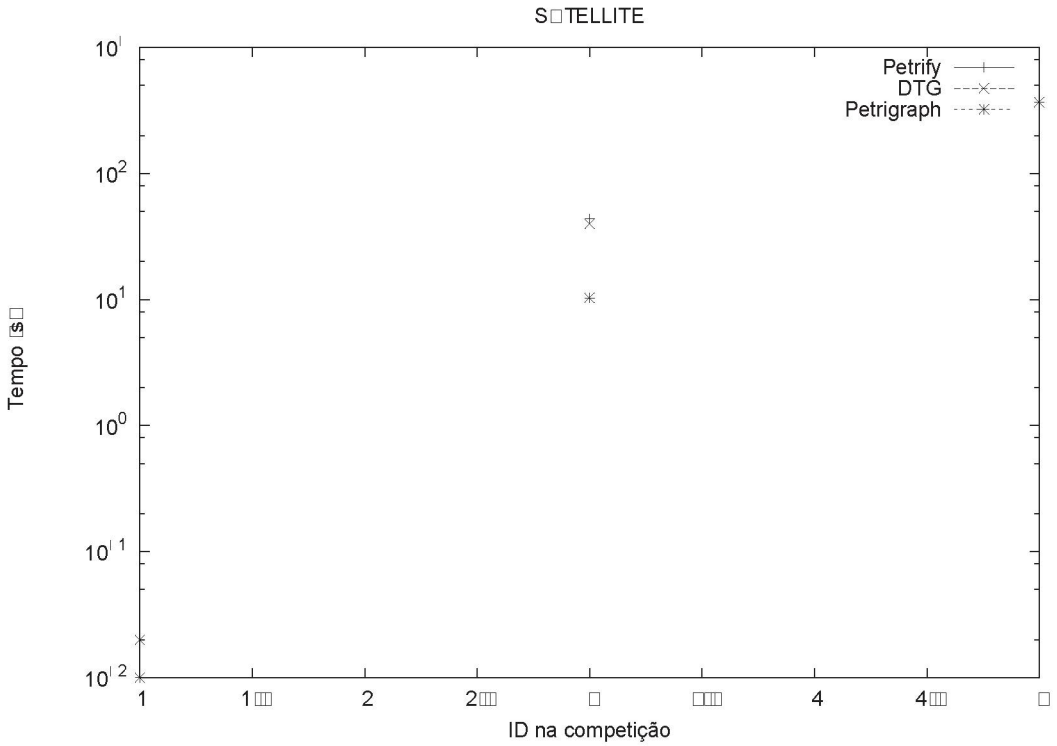


Figura 5.22: Tempo de execução para o domínio SATELLITE

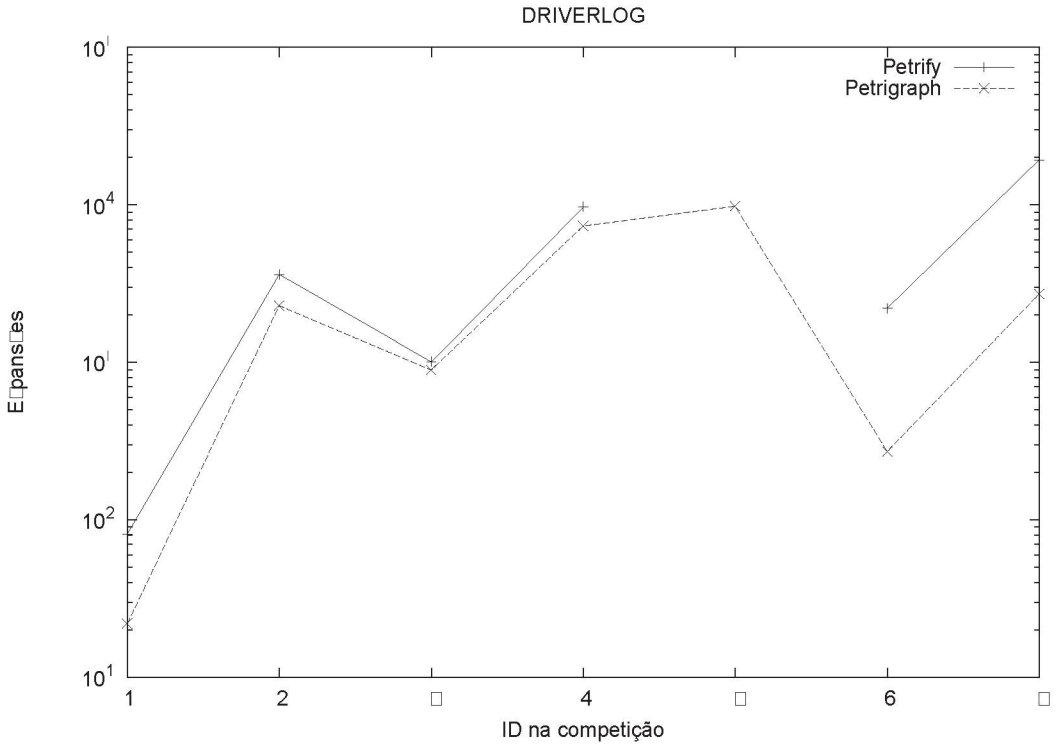


Figura 5.23: Expansões para o domínio DriverLog

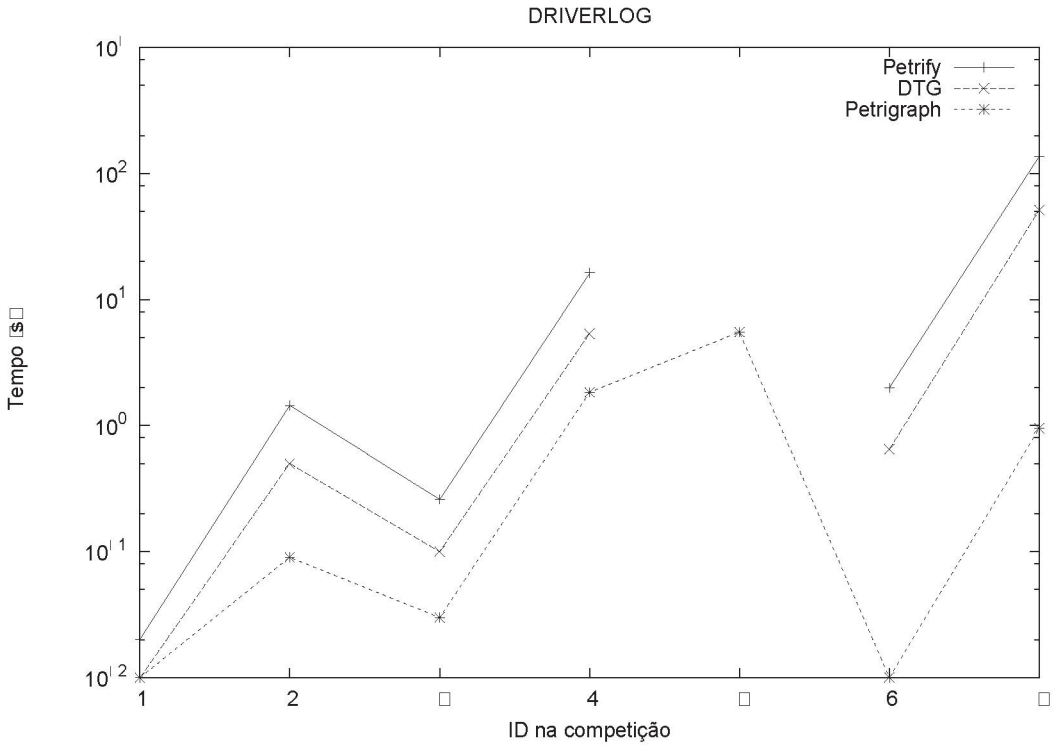


Figura 5.24: Tempo de execução para o domínio DriverLog

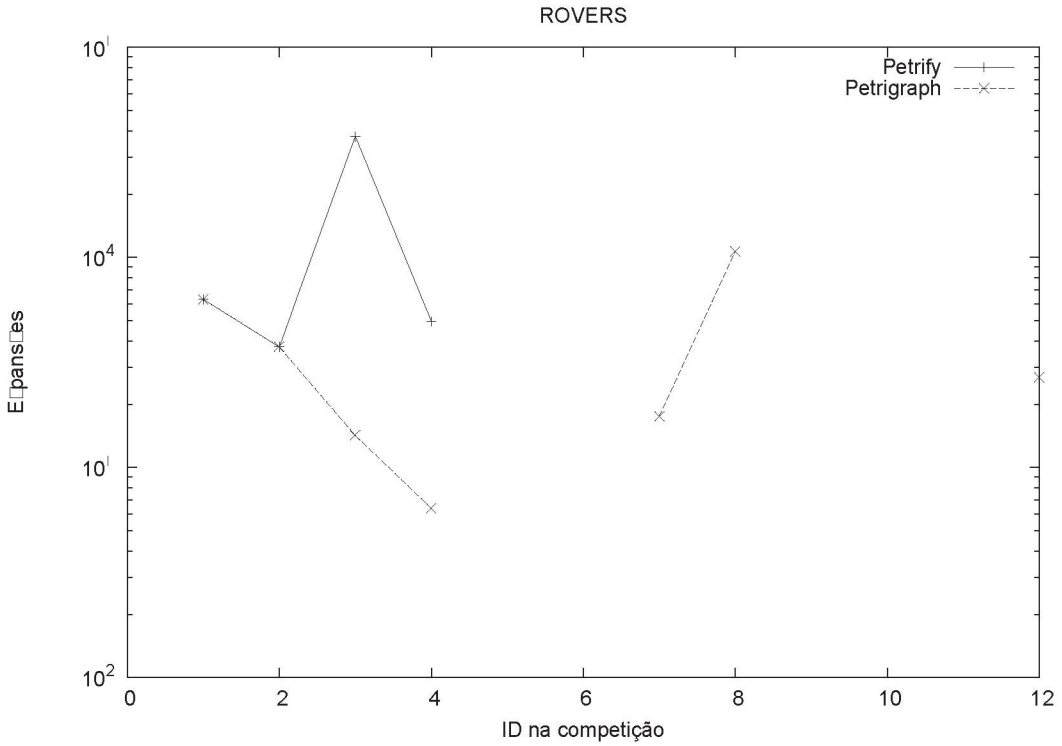


Figura 5.25: Expansões para o domínio Rovers

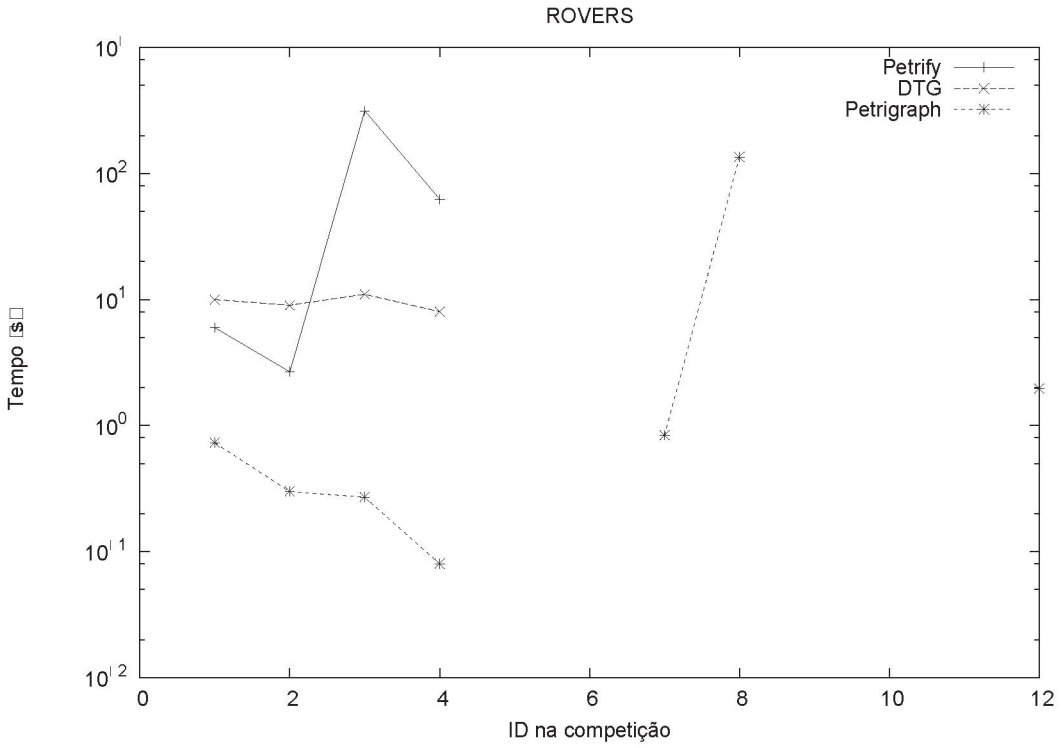


Figura 5.26: Tempo de execução para o domínio Rovers

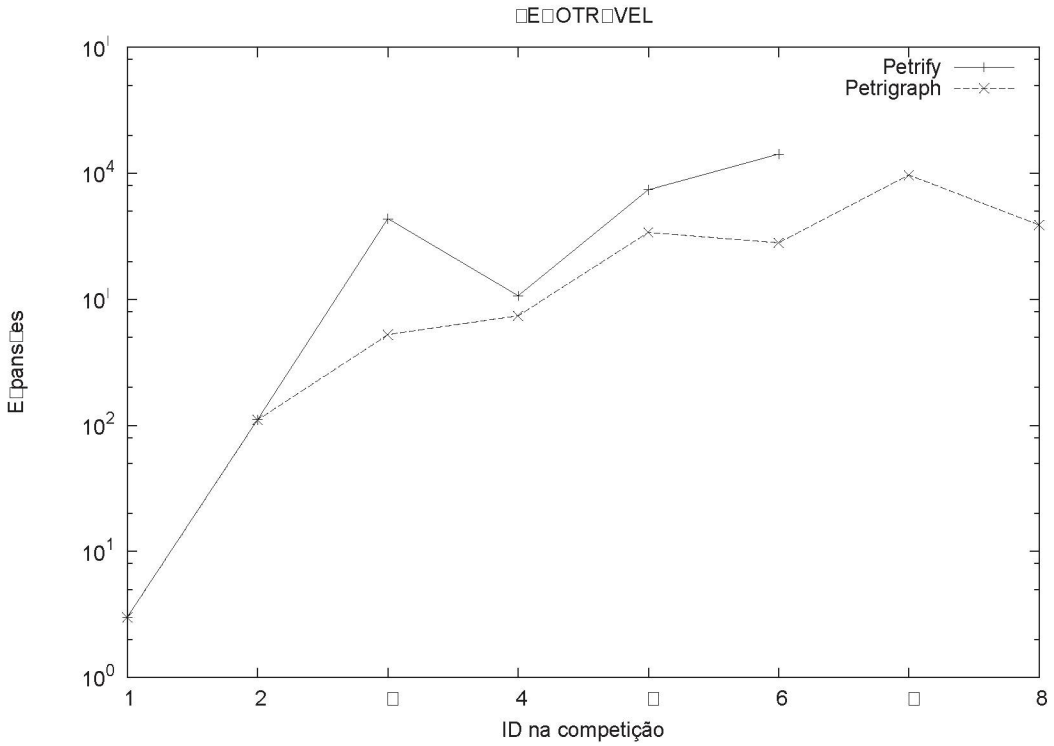


Figura 5.27: Expansões para o domínio ZenoTravel

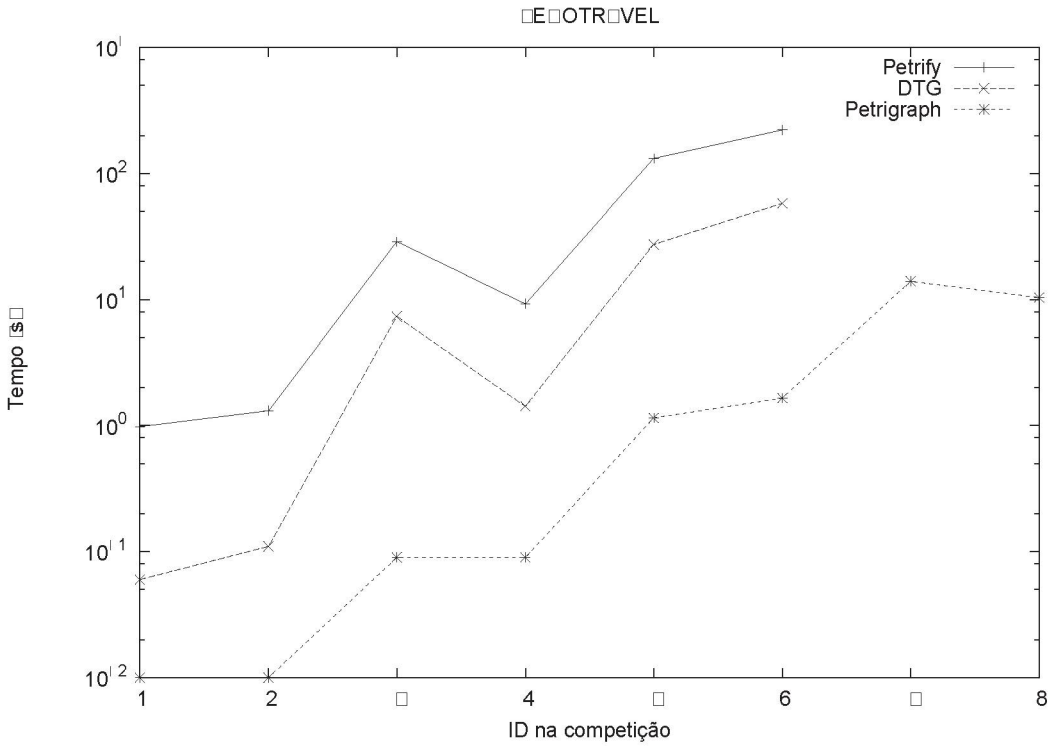


Figura 5.28: Tempo de execução para o domínio ZenoTravel

5.3.3 Expansões e tempo de execução, heurística h^1

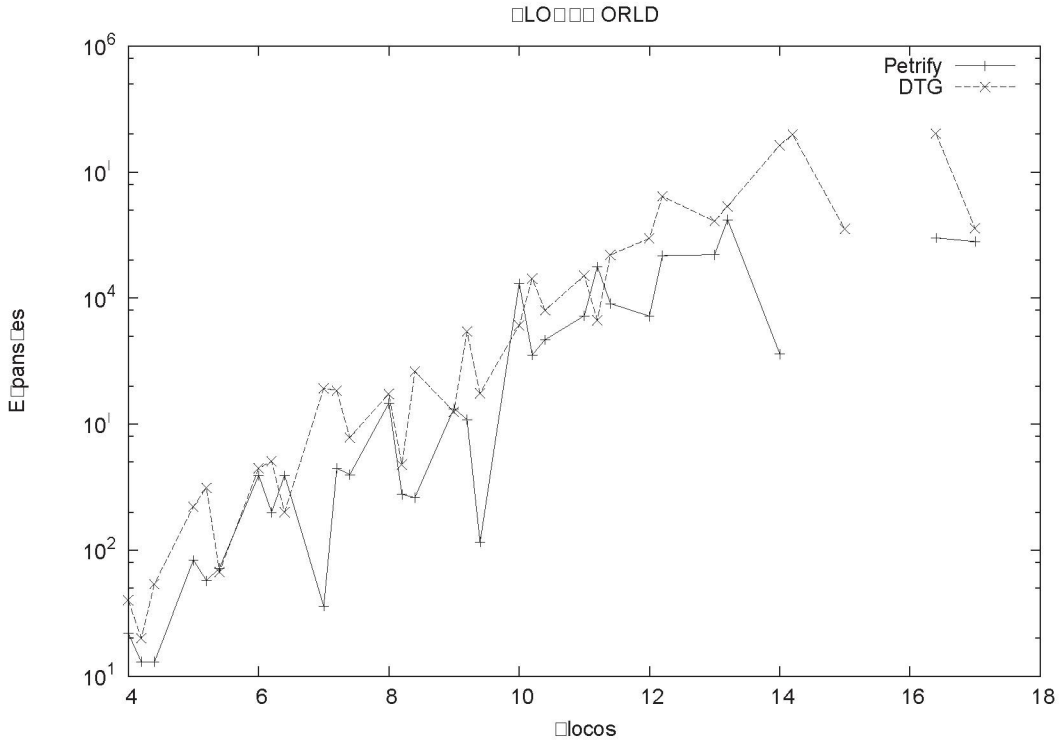


Figura 5.29: Expansões para o domínio BlocksWorld

O número de expansões do BlocksWorld pelo *Petrigraph* é ligeiramente maior, mas o tamanho menor da rede implica num tempo ligeiramente mais rápido para a maioria dos problemas. Como no tamanho da rede, o emprego de DTGs é o fator que influencia a variação.

O *Petrigraph* tem uma influência massiva nos problemas do domínio Logistics, aumentando a velocidade de resolução em mais de 100 vezes na maioria dos casos - exceto pelo primeiro caso com 7 predicados de objetivo, onde o uso de DTGs atrasa consideravelmente a resolução.

A retirada de constantes aumenta a velocidade no domínio Elevator, mas o uso de DTGs e corte aumenta o número de expansões e torna a resolução mais lenta e errática em termos de tempo. Durante os testes, verificou-se que isso pode ter ocorrido devido à reestruturação da heurística com a remoção de caminhos impossíveis - a heurística tornou-se mais correta em relação ao progresso de cada objetivo (localização do elevador e passageiros servidos), mas não existe um controle da ordem na qual os objetivos devem

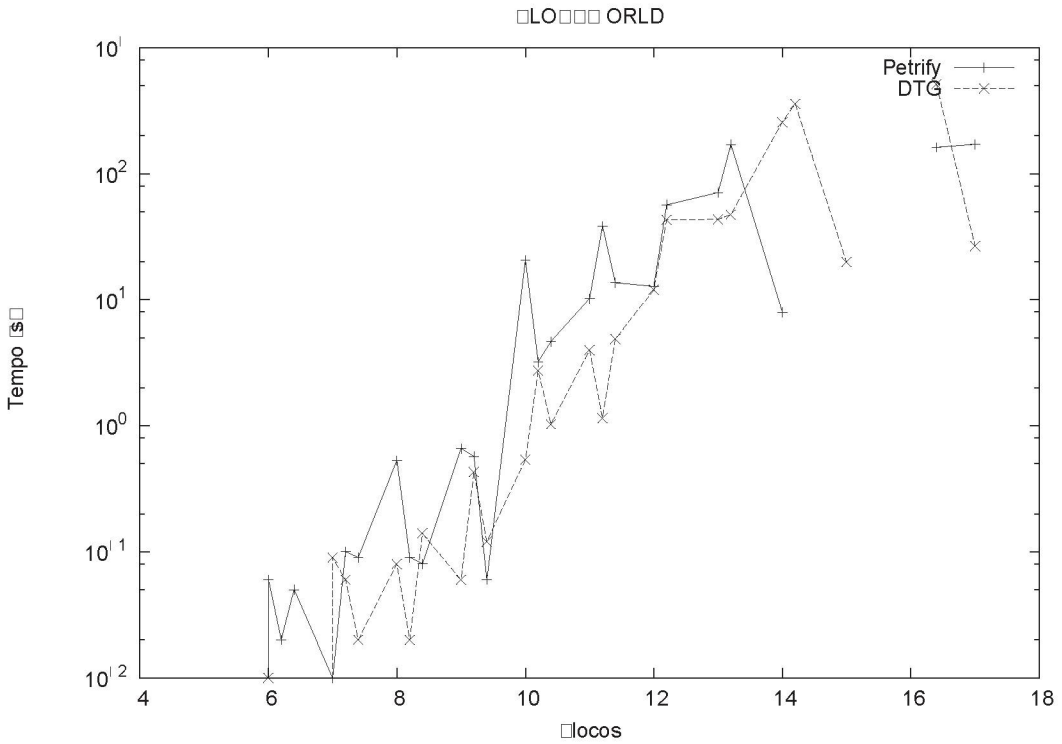


Figura 5.30: Tempo de execução para o domínio BLOCKSWORLD

ser preenchidos (os passageiros devem estar no elevador antes que ele siga para o andar de saída).

O domínio DRIVERLOG beneficia-se tanto do uso de DTGs quanto remoção de constantes. O número de expansões do desdobramento é reduzido quando as constantes são removidas, e o tempo de processamento do *Petrigraph* chega a ser 1000 vezes mais rápido que o do *Petrify*.

No domínio ROVERS o número de expansões e tempo de execução foram extremamente variáveis dependendo do problema. Enquanto o *Petrigraph* mostrou-se mais eficiente nos casos menores, em vários dos casos maiores o desdobramento para o *Petrigraph* ultrapassou o limite de tempo, mesmo quando este limite foi estendido para 20 minutos, e não foi computado. Não é possível determinar o comportamento generalizado do método a partir destes casos.

No domínio SATELLITE, apenas cinco problemas foram resolvidos no limite de tempo, mesmo quando este limite foi estendido para 20 minutos. Nos cinco problemas resolvidos, o *Petrigraph* mostrou uma melhora no tempo de execução, cujo fator varia para cada um

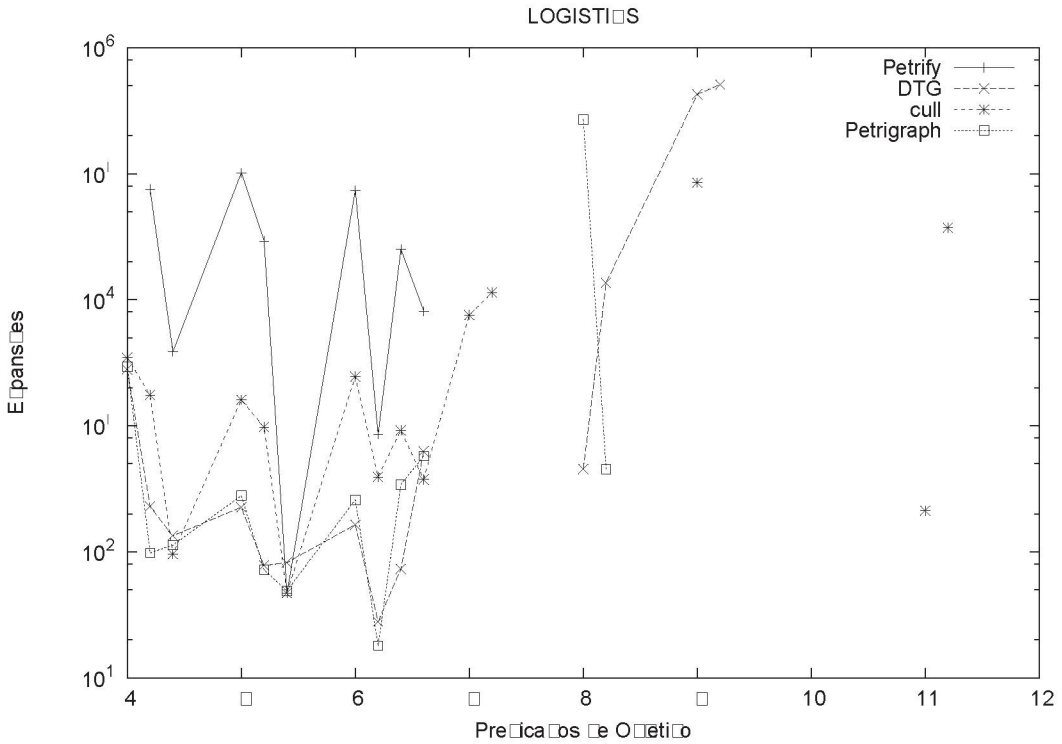


Figura 5.31: Expansões para o domínio LOGISTICS

dos problemas.

Tanto o número de expansões quanto o tempo de resolução dos problemas do domínio ZENOTRAVEL se reduziram com a adição de passos do *Petrigraph* - DTG, DTG mais corte, e todos os três passos.

Em geral, o método de criação de DTGs teve sucesso mais definitivo nos casos onde a equivalência entre variáveis de estado booleanas e multivaloradas é bem marcada. É possível que o fato da rede construída ser mais irregular, ou seja, conter apenas transições possíveis no problema específico ao invés de todas as transições teoricamente possíveis no domínio, cause alterações prejucidicais à heurística h^1 em alguns domínios.

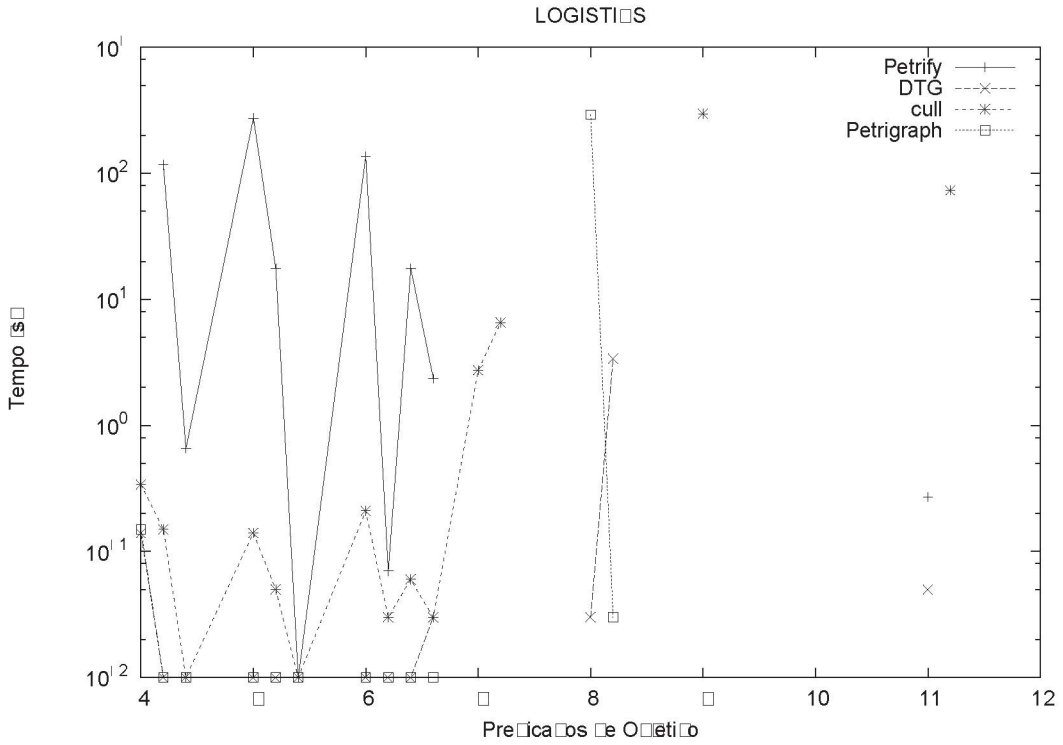


Figura 5.32: Tempo de execução para o domínio Logistics

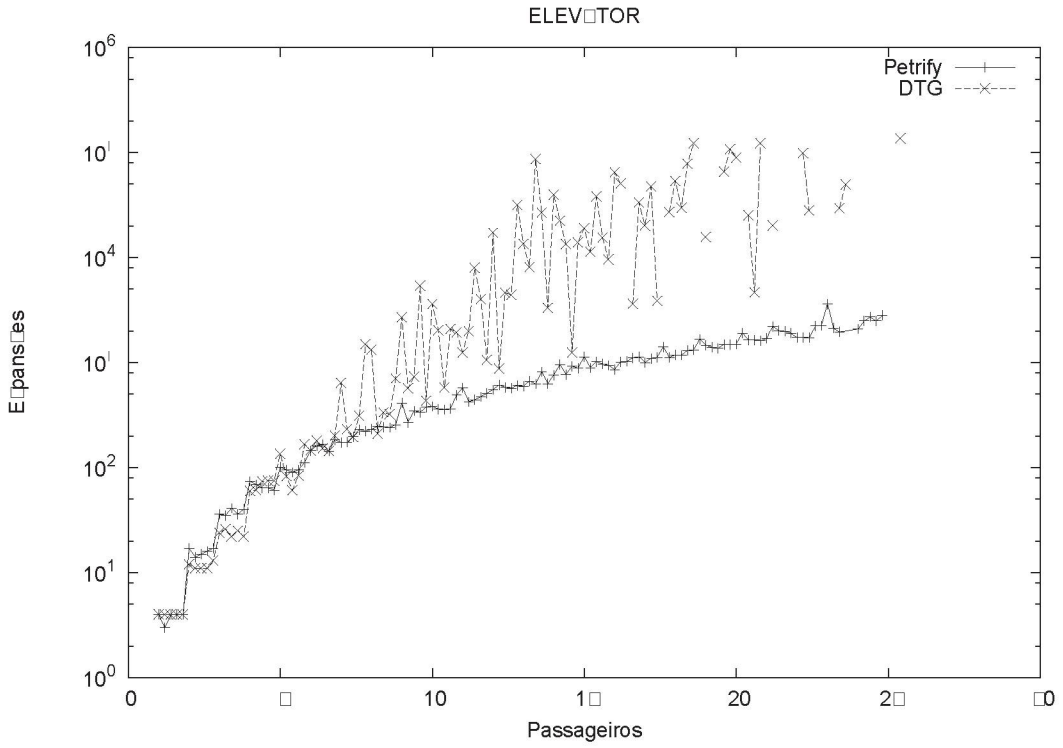


Figura 5.33: Expansões para o domínio Elevator

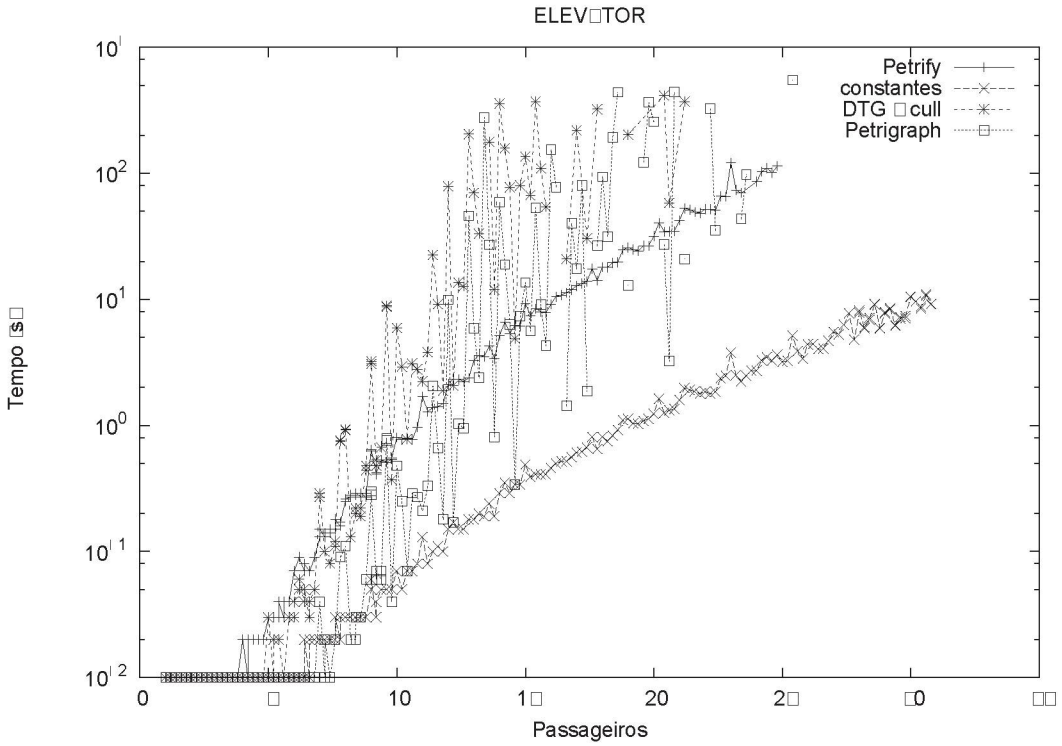


Figura 5.34: Tempo de execução para o domínio ELEVATOR

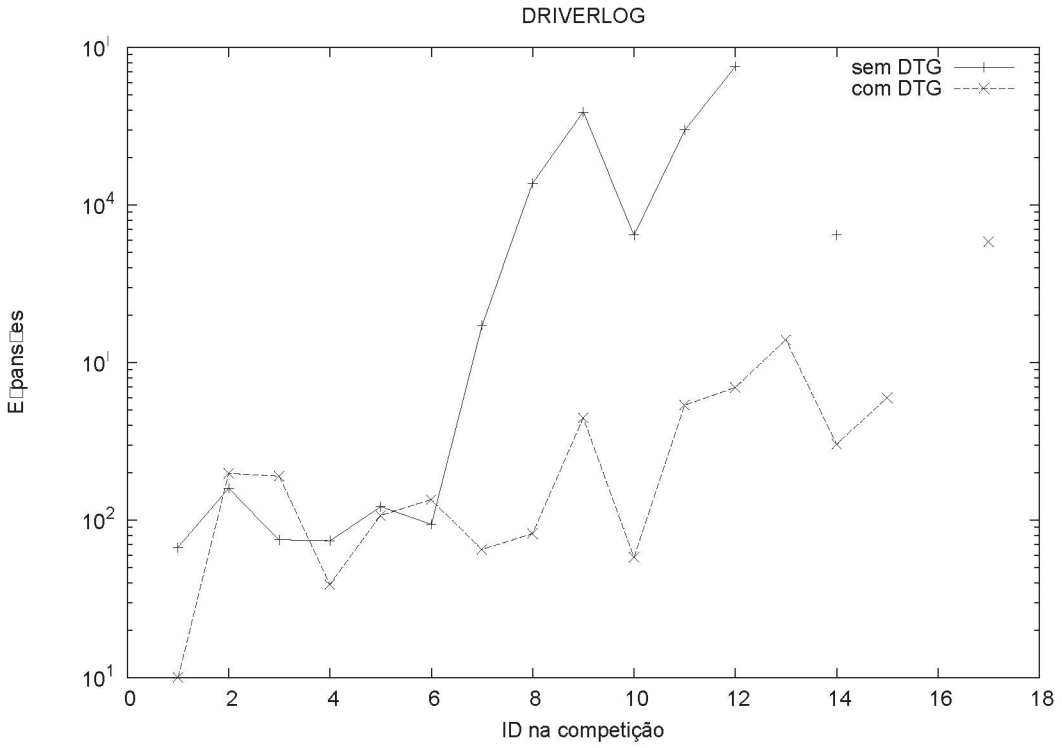


Figura 5.35: Expansões para o domínio DRIVERLOG

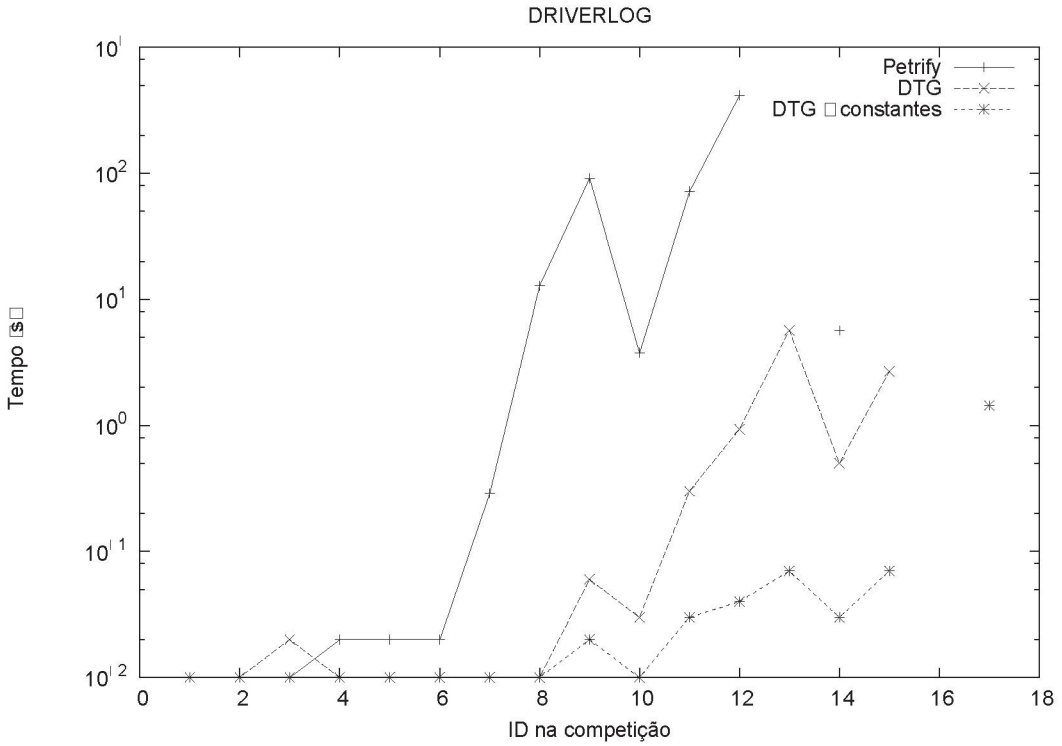


Figura 5.36: Tempo de execução para o domínio DriverLog

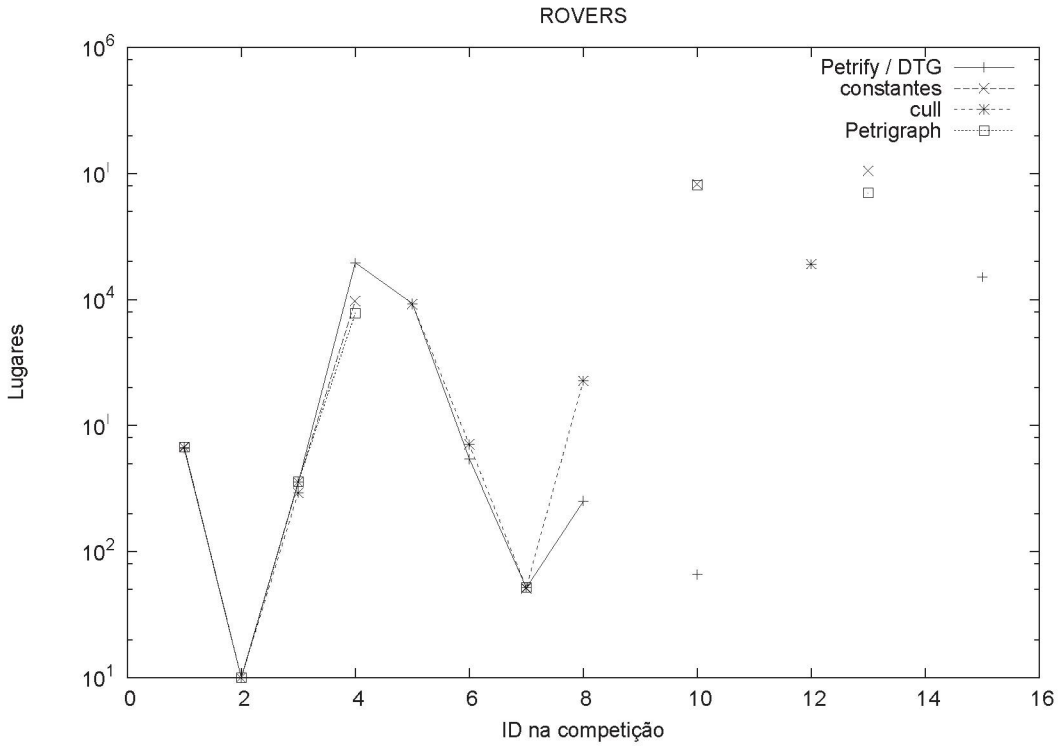


Figura 5.37: Expansões para o domínio Rovers

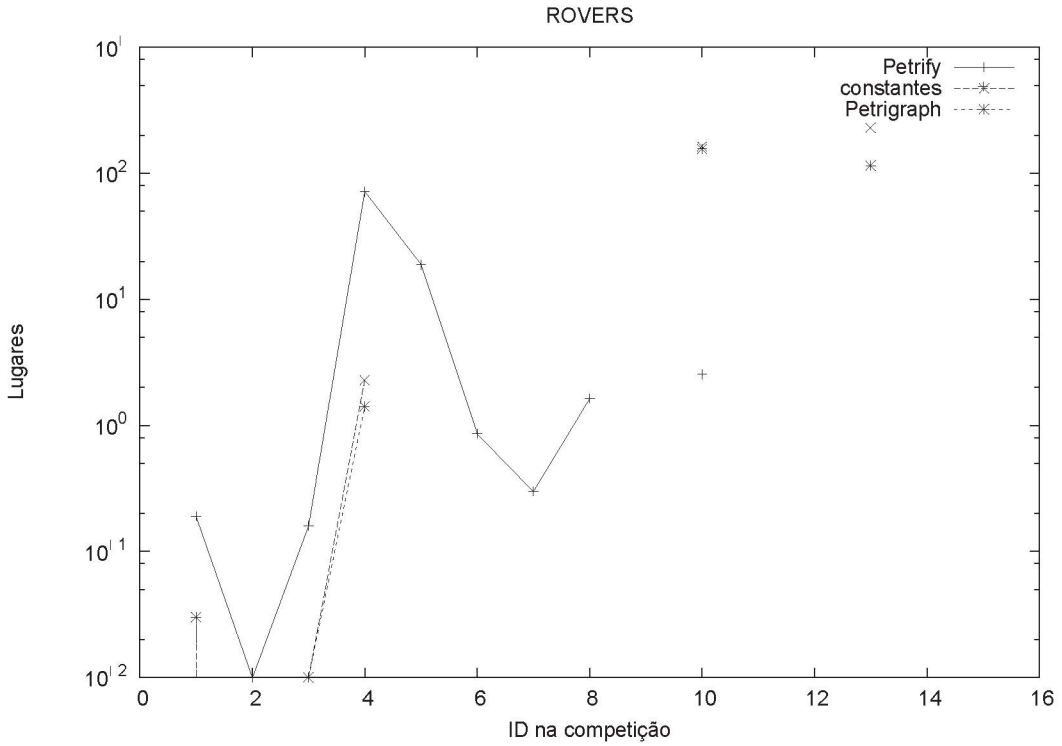


Figura 5.38: Tempo de execução para o domínio ROVERS

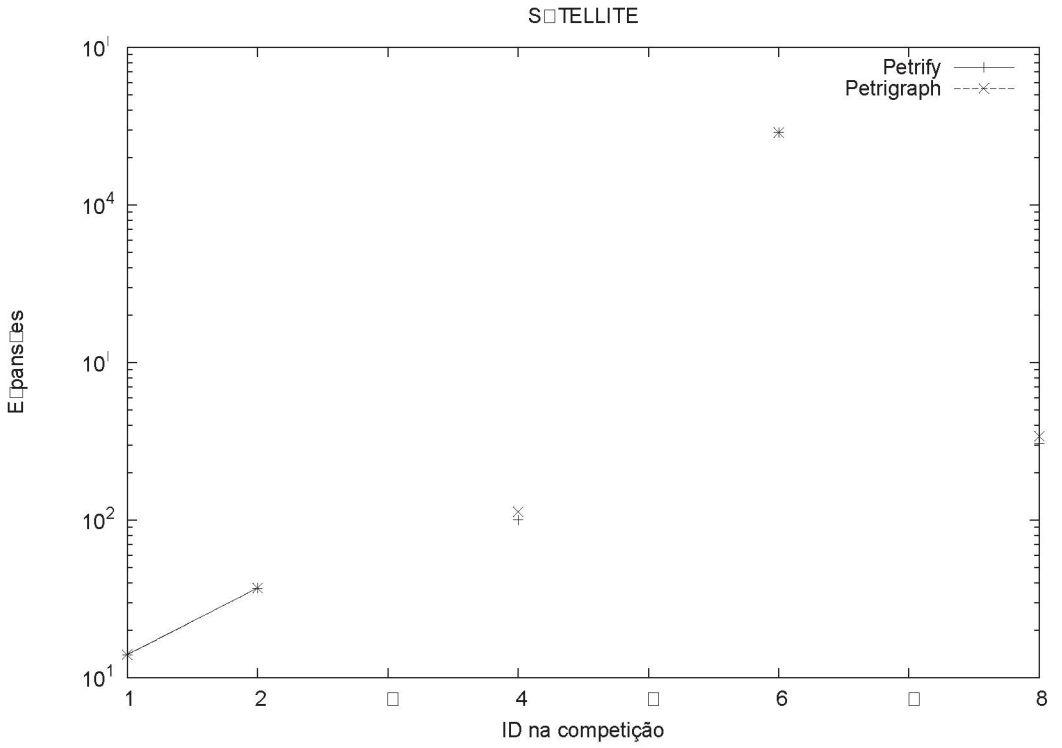


Figura 5.39: Expansões para o domínio SATELLITE

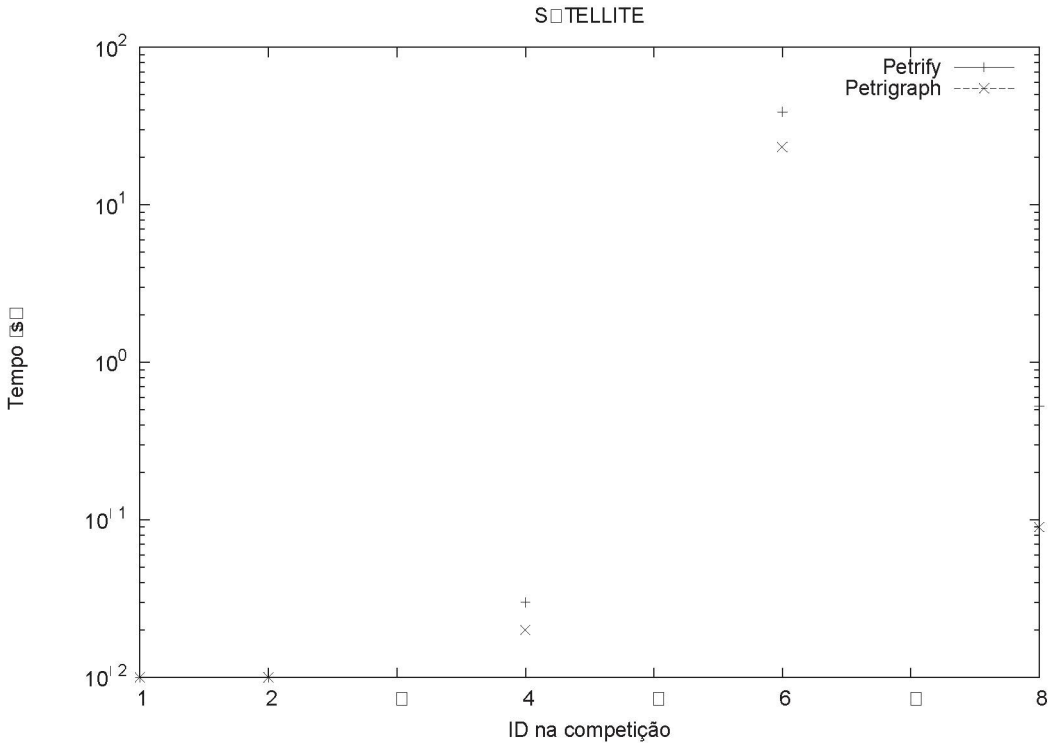


Figura 5.40: Tempo de execução para o domínio SATELLITE

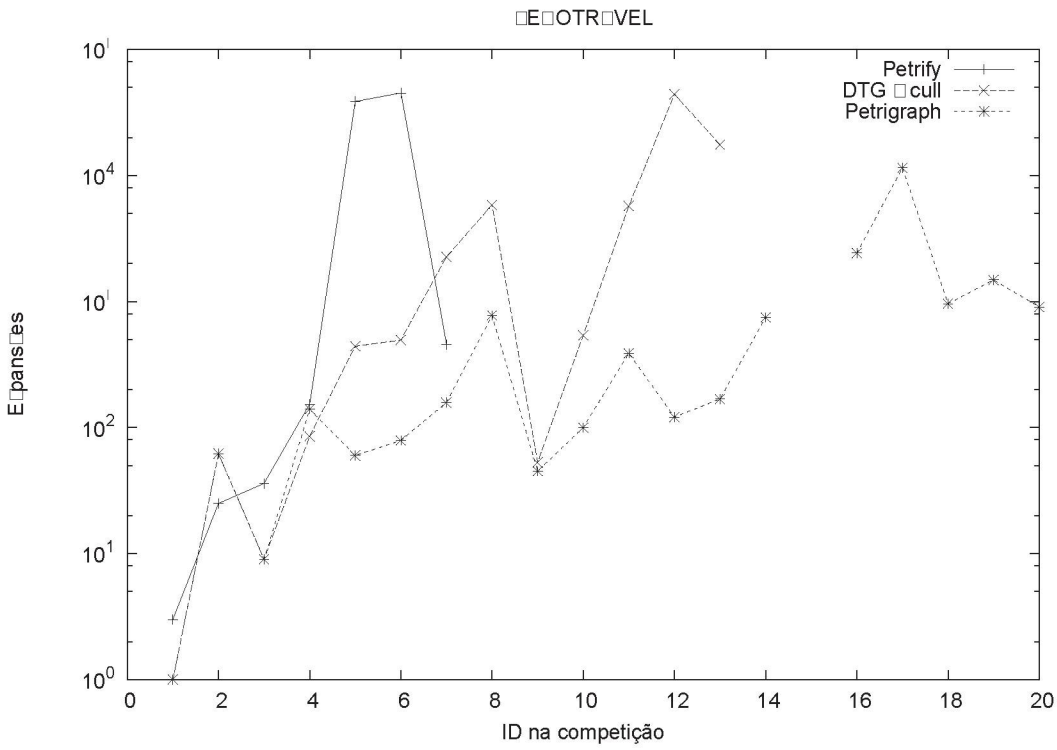


Figura 5.41: Expansões para o domínio ZENOTRAVEL

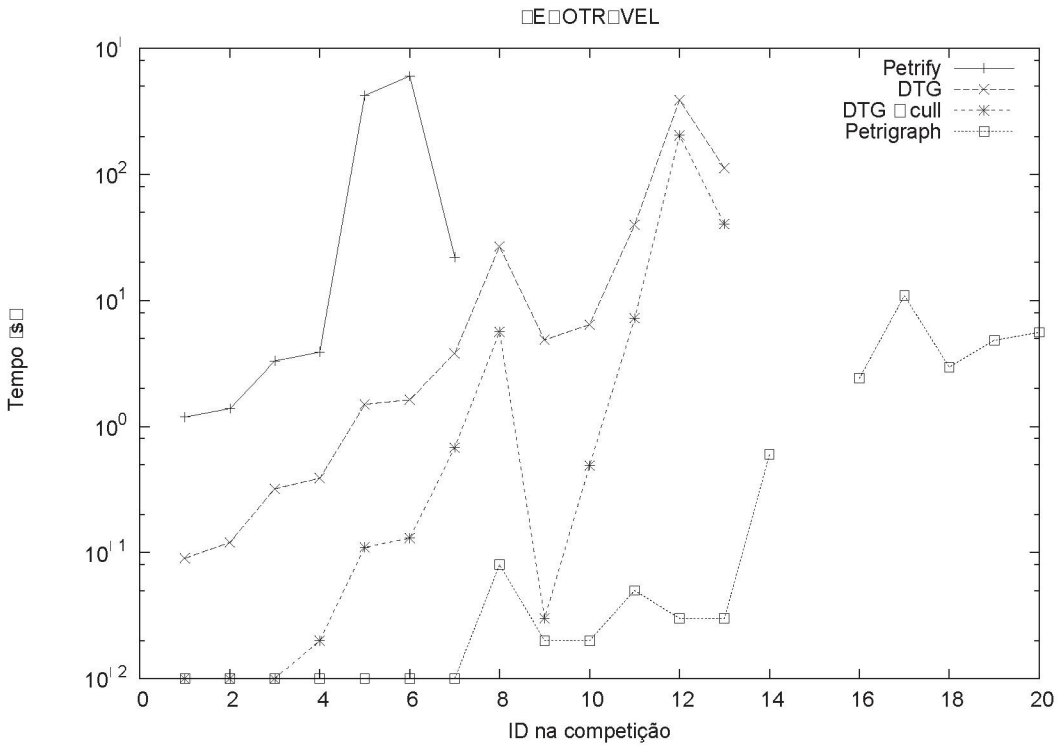


Figura 5.42: Tempo de execução para o domínio ZenoTravel

5.3.4 Comparação de tamanho de objetivo e tempo, incluindo SatPlan

Foi levado em conta o tamanho total do plano, sem considerar ações concorrentes.

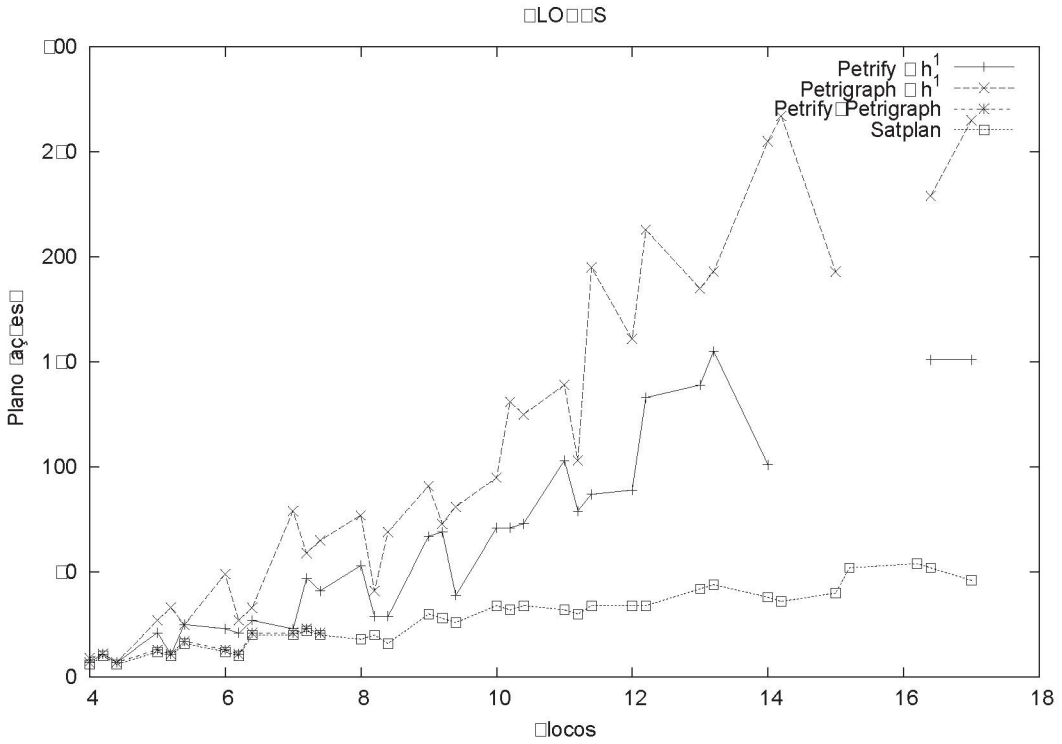


Figura 5.43: Comparação de tamanho de plano para o domínio BlocksWorld

O *Satplan* leva mais tempo que o *Petrify* e *Petrigraph* com heurística h^1 para problemas de tamanho menor, mas torna-se relativamente mais rápido conforme o tamanho do problema aumenta. Utilizando a heurística h^1, o *Petrigraph* gera planos de tamanho maior que o *Petrify* no domínio BlocksWorld, com um tempo quase sempre ligeiramente menor de execução e ambos geram planos de tamanho muito maior que o mínimo. Sem a heurística h^1, o plano mínimo é encontrado, mas com tempo superior em várias ordens de magnitude ao do *Satplan*.

O *Satplan* não conseguiu resolver o domínio Logistics como fornecido, alegando erros na sintaxe. O *Petrigraph* com heurístics h^1 gerou planos de tamanho bem próximo ao mínimo, com uma grande vantagem de tempo em relação ao *Petrify* com heurística e os métodos sem heurística.

O uso de DTGs no domínio Elevator diminui sensivelmente o tamanho do plano

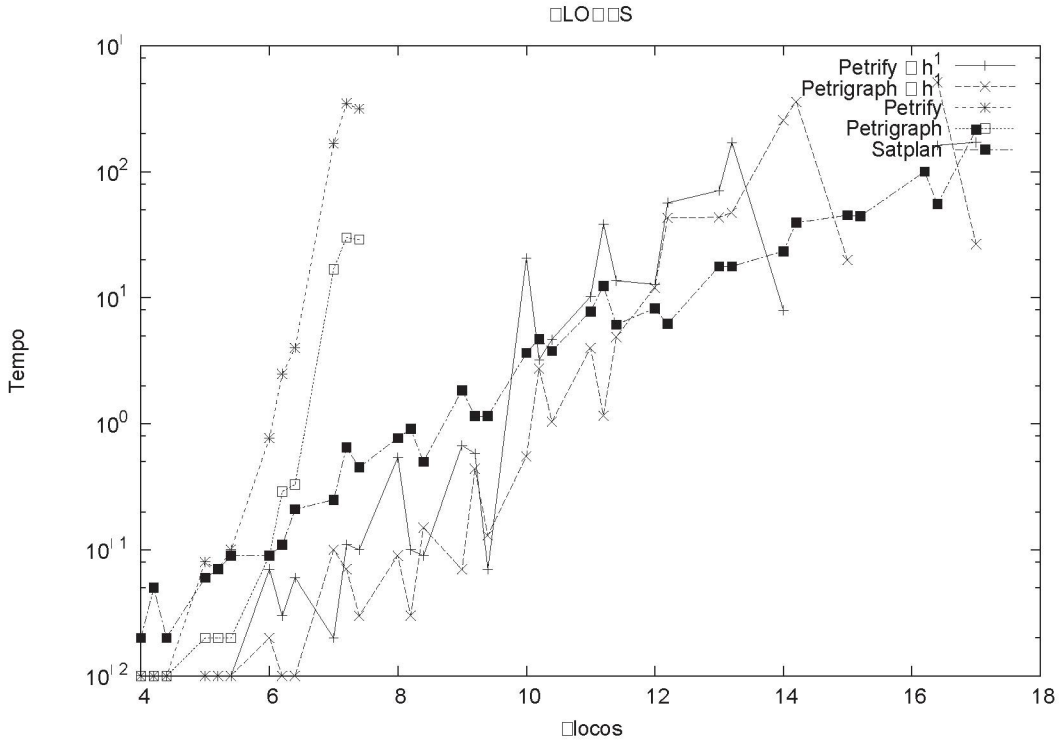


Figura 5.44: Comparação de tempo de execução para o domínio BLOCKSWORLD

encontrado usando a heurística h^1. A causa dessa redução pode ser a mesma reestruturação da heurística que aumentou o número de expansões necessárias para um plano, forçando o plano a ser mais acurado. O tempo de execução com a heurística h^1 é muito menor que o do Satplan, com um aumento no tamanho do plano relativamente pequeno. A execução sem a heurística encontra o plano mínimo em um tempo ligeiramente maior que o do Satplan.

No domínio DRIVERLOG, a resolução sem o uso de heurística é significantemente mais rápida no *Petrigraph* do que no *Petrify*. Com o uso da heurística h^1, o *Petrigraph* foi capaz de encontrar planos de tamanho próximo ao ótimo em tempos muito pequenos comparados tanto com o *Petrify* quanto com o *Satplan*.

No domínio ROVERS, o desempenho do *Petrigraph* foi equivalente em termos de tamanho de plano com o *Petrify*, com uma redução de tempo média de 10 vezes, tanto nos casos sem heurística quanto nos casos com heurística h^1. No entanto, o *Satplan* resolveu os problemas com velocidade maior e com tamanho mínimo.

Os resultados do domínio SATELLITE, tanto do *Petrify* quanto do *Petrigraph*, foram

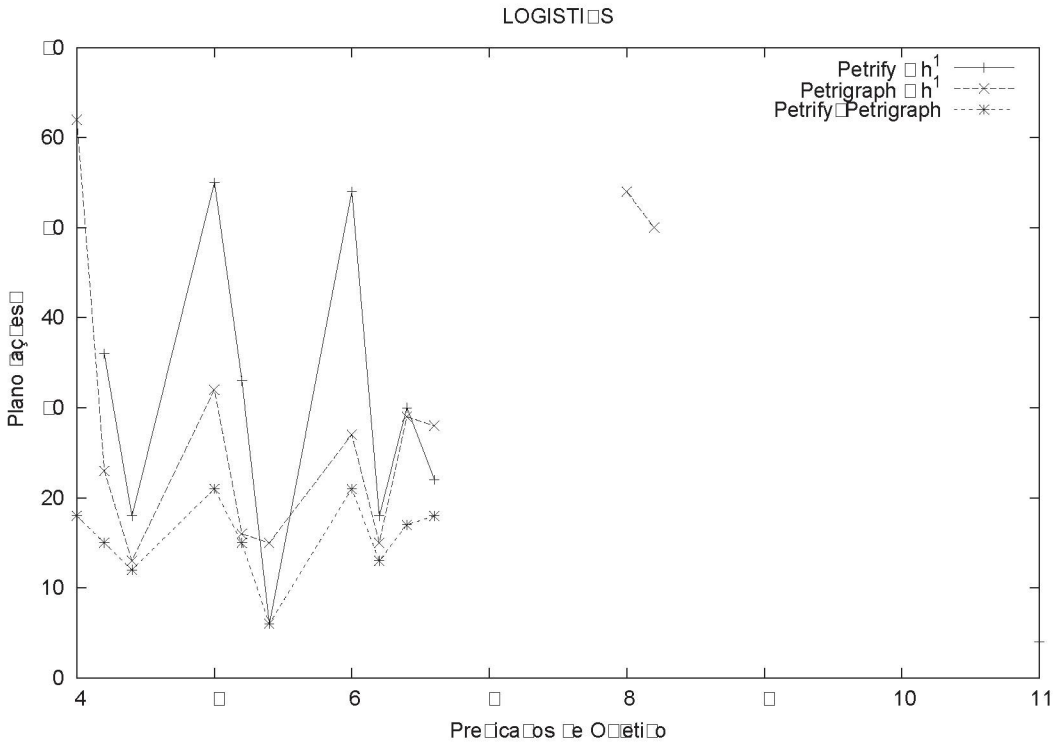


Figura 5.45: Comparação de tamanho de plano para o domínio LOGISTICS

extremamente inconstantes em relação aos do *Satplan*. O *Petrigraph* demonstrou uma pequena vantagem sobre o *Petrify* nos problemas que ambos conseguiram resolver no limite de tempo, mas a performance de ambos relativa ao *Petrify* não pôde ser generalizada.

No domínio ZENOTRAVEL o *Petrigraph* conseguiu planos de tamanho próximo aos dos encontrados pelo *Petrify*, em um tempo bastante menor, tanto na versão sem heurística quanto na versão com heurística h^1. O *Petrigraph* com heurística h^1 teve uma grande vantagem em performance sobre o *Satplan*, e encontrou planos bastante próximos dos planos ótimos conforme encontrados pela busca em largura.

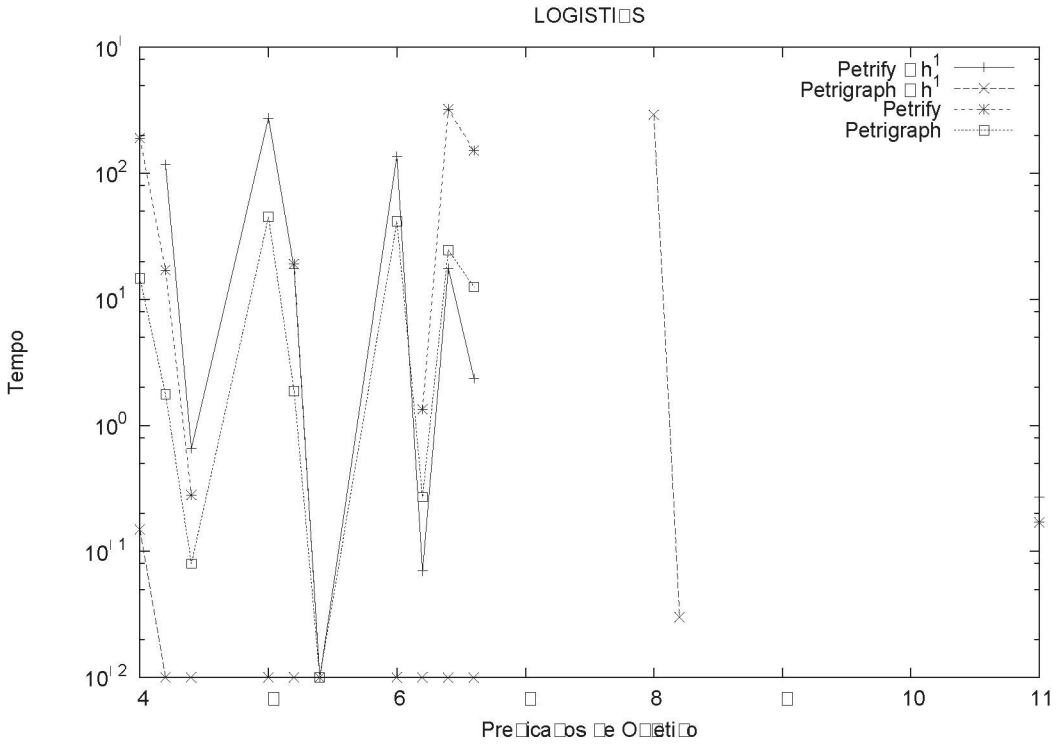


Figura 5.46: Comparação de tempo de execução para o domínio LOGISTICS

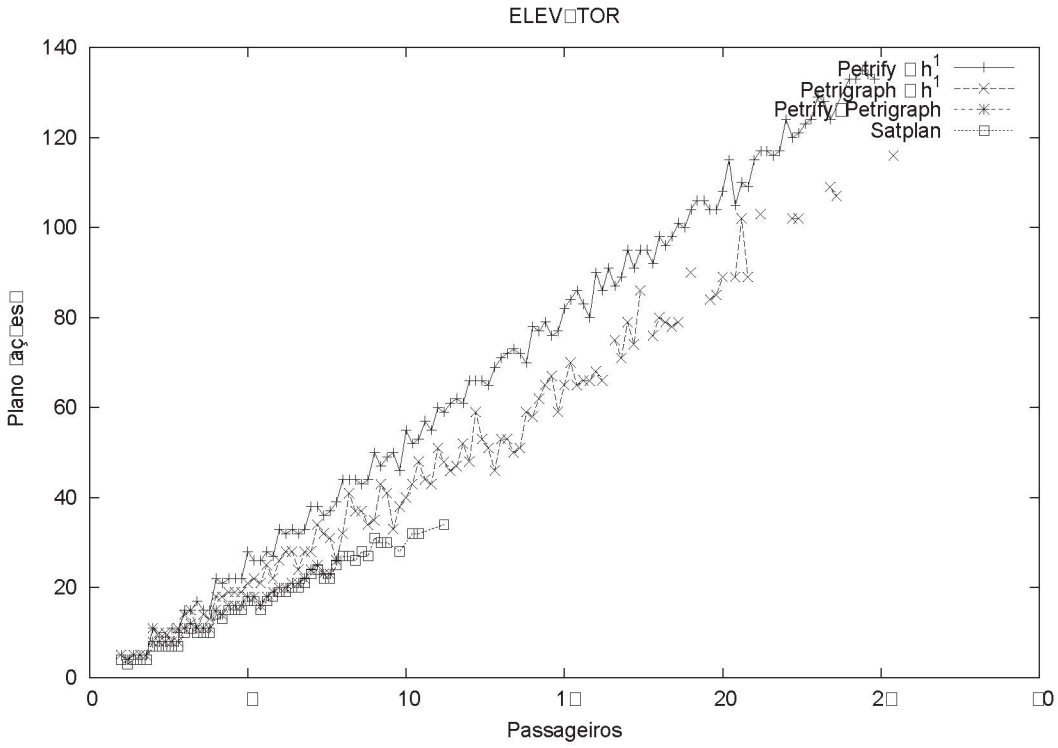


Figura 5.47: Comparação de tamanho de plano para o domínio ELEVATOR

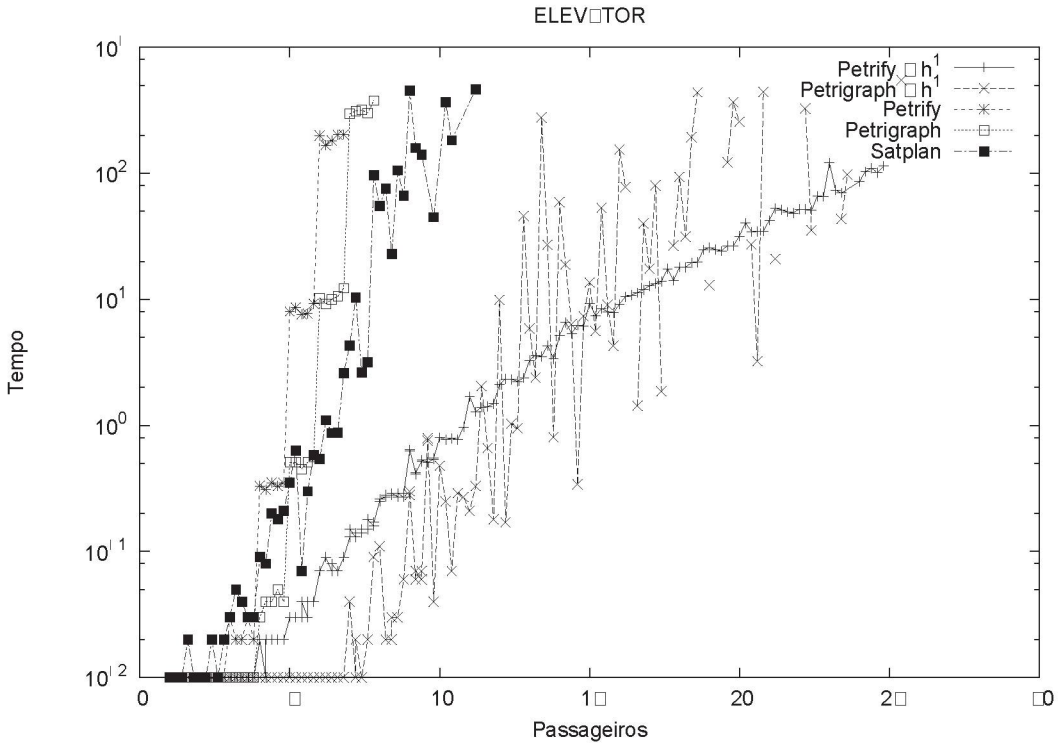


Figura 5.48: Comparação de tempo de execução para o domínio ELEVATOR

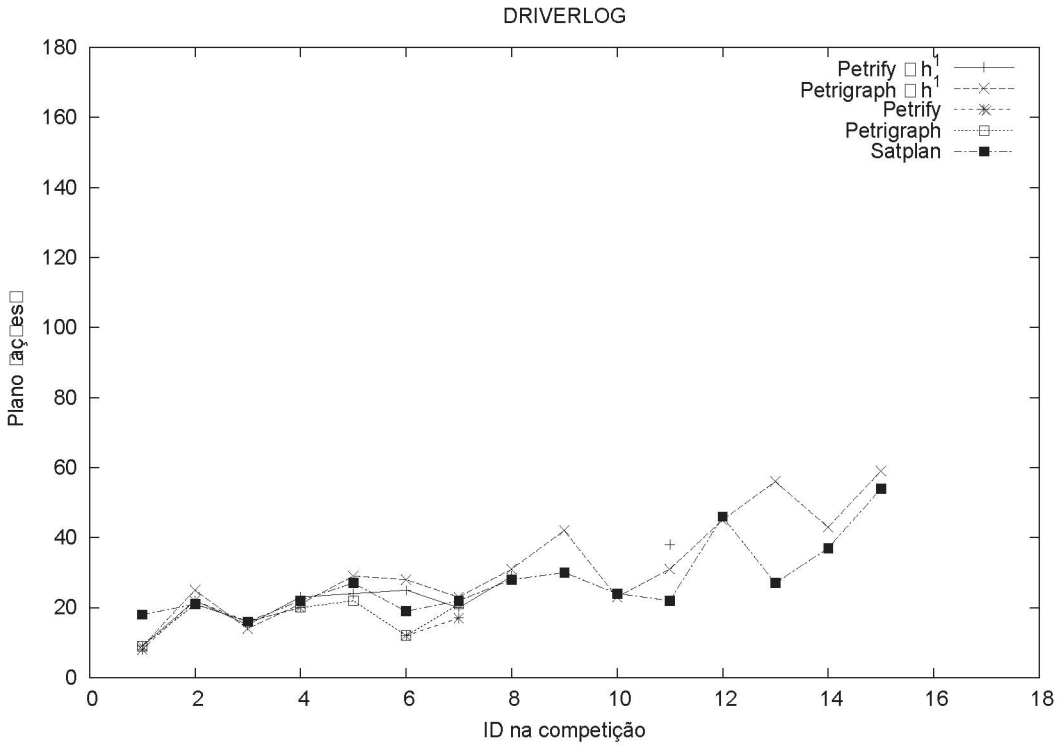


Figura 5.49: Comparação de tamanho de plano para o domínio DRIVERLOG

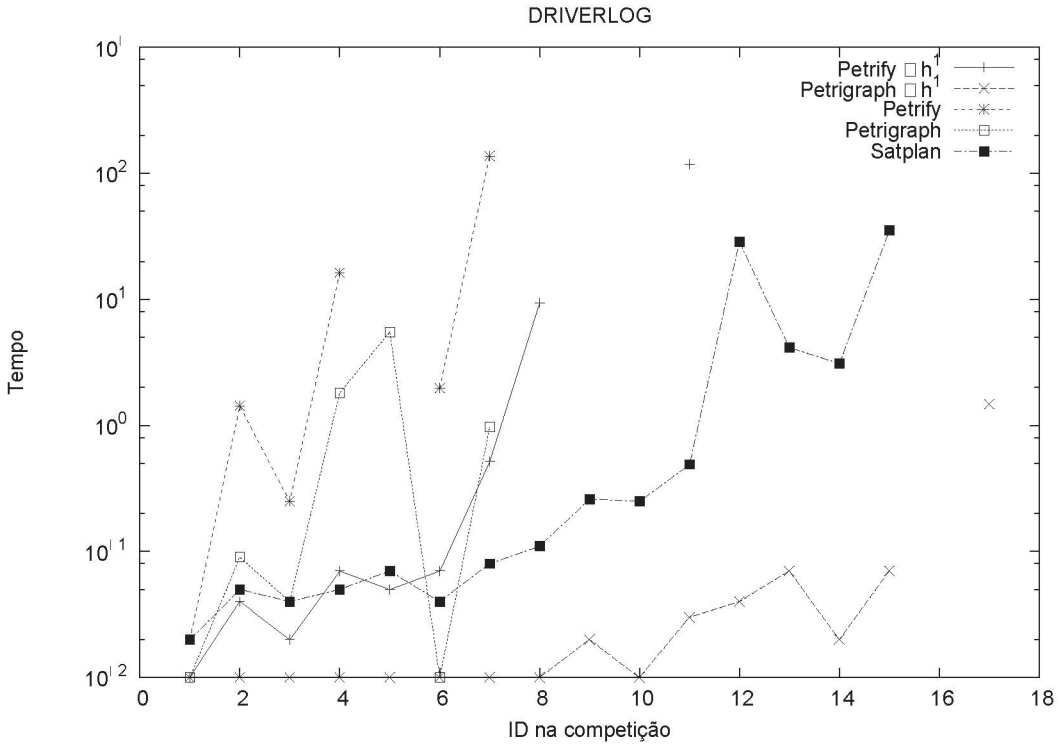


Figura 5.50: Comparação de tempo de execução para o domínio DriverLog

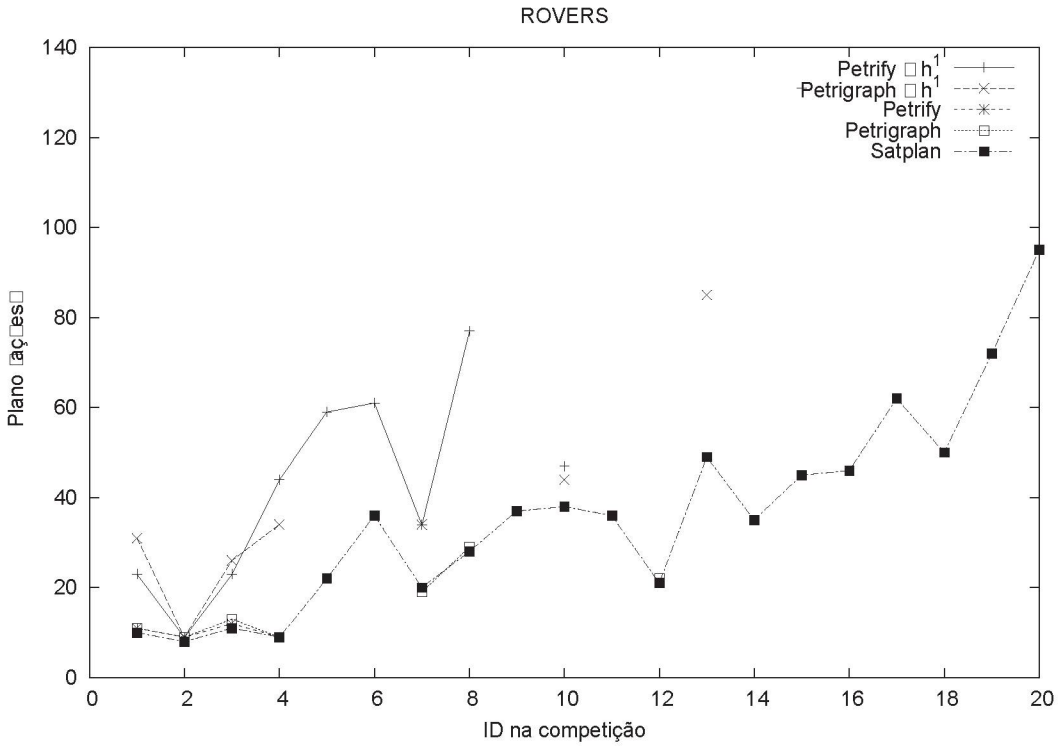


Figura 5.51: Comparação de tamanho de plano para o domínio Rovers

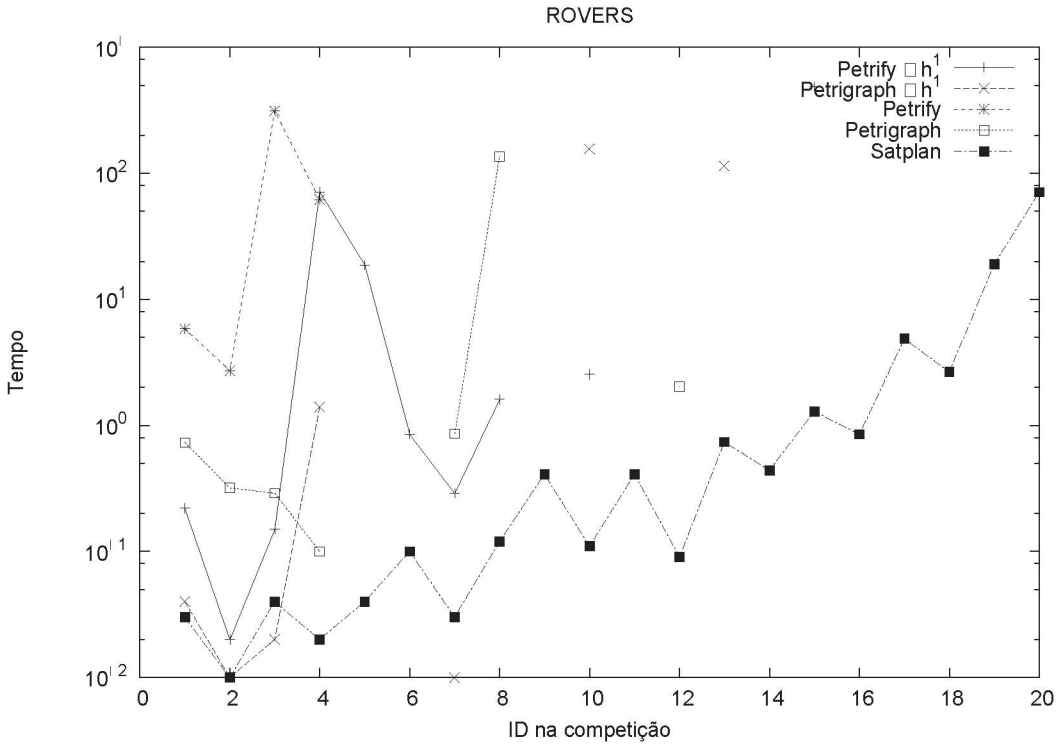


Figura 5.52: Comparação de tempo de execução para o domínio ROVERS

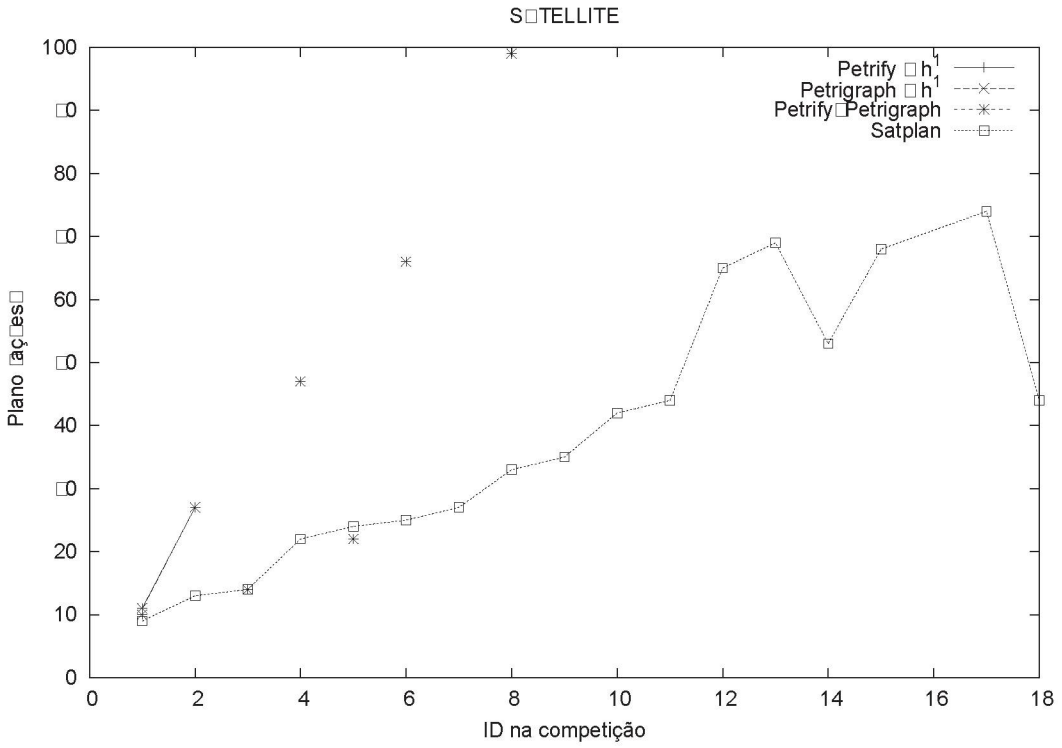


Figura 5.53: Comparação de tamanho de plano para o domínio SATELLITE

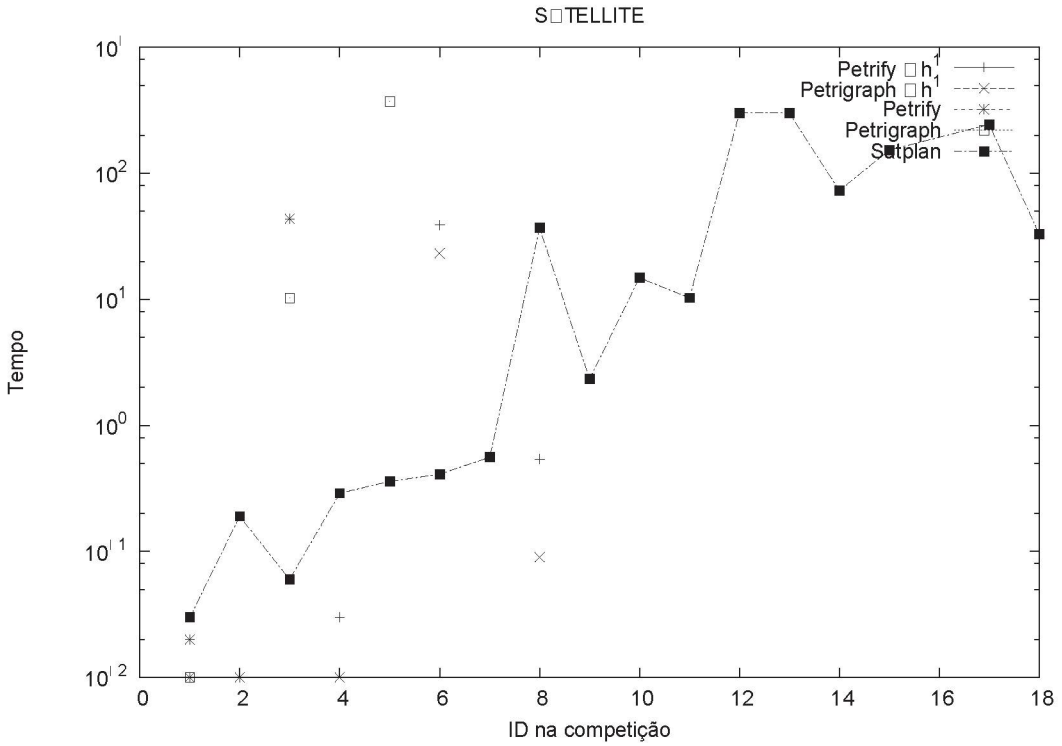


Figura 5.54: Comparação de tempo de execução para o domínio SATELLITE

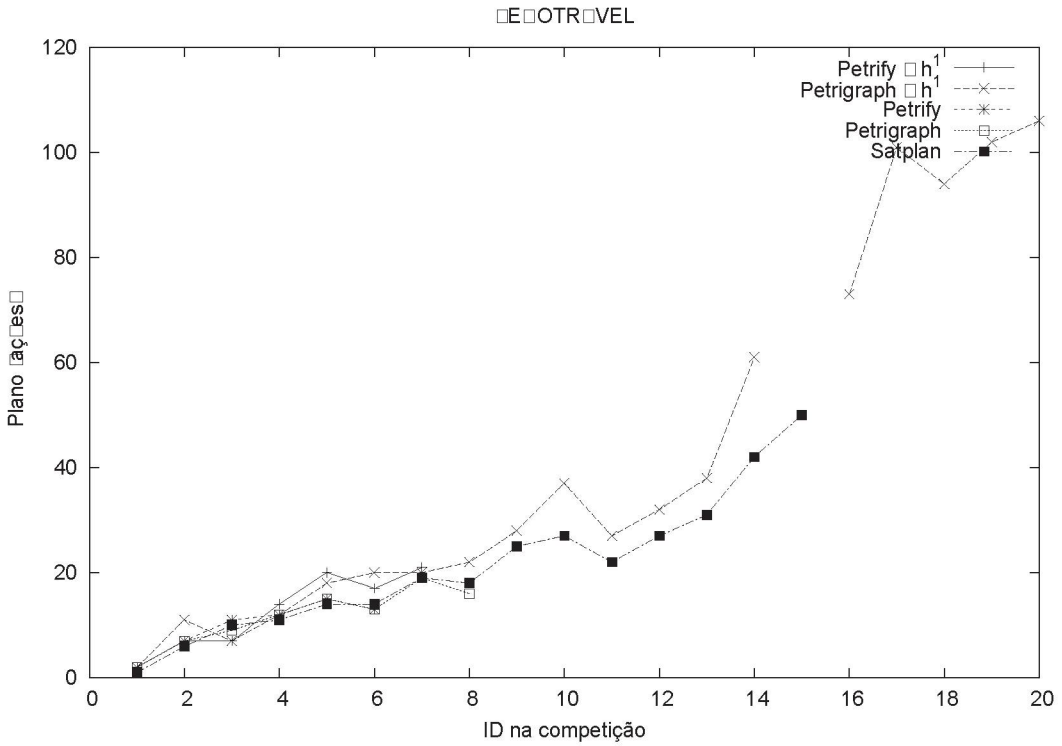


Figura 5.55: Comparação de tamanho de plano para o domínio ZENOTRAVEL

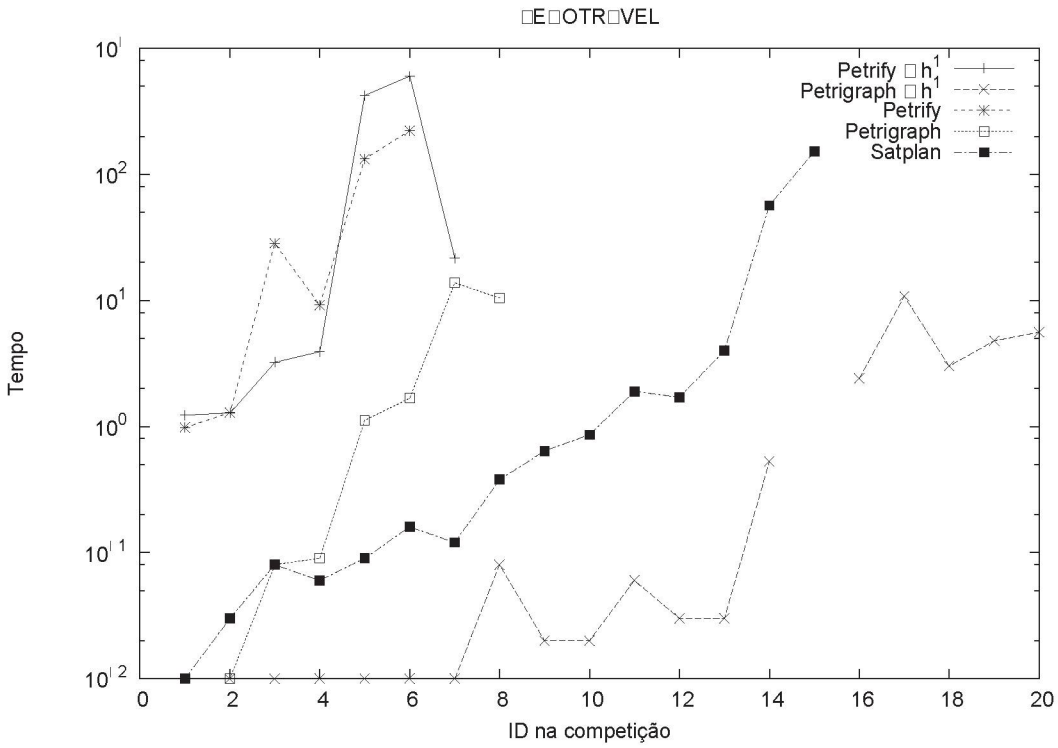


Figura 5.56: Comparação de tempo de execução para o domínio ZenoTravel

CAPÍTULO 6

CONCLUSÕES E TRABALHOS FUTUROS

Foram realizados testes para medição da eficácia do método *Petrigraph* quando comparado ao *Petrify*, e foi demonstrado que todas as etapas que diferenciam o *Petrigraph* do *Petrify* têm um diferencial no tamanho da rede e no tempo. Em termos de tamanho, a rede de Petri gerada pelo *Petrigraph* sempre é igual ou menor à gerada pelo *Petrify*, e em alguns dos casos testados é 10 ou 100 vezes menor. Para problemas maiores, a vantagem deve continuar aumentando, pela eliminação de multiplicações desnecessárias de transições.

Em relação ao tempo, na busca em largura, o *Petrigraph* mostrou-se consistentemente superior ao *Petrify*, enquanto que nas buscas com heurística, o uso de DTGs foi mais eficiente em certos domínios do que em outros. Os ganhos de velocidade nos domínios em que o *Petrigraph* foi mais eficiente mudam de domínio para domínio, variando entre 10 vezes e 1000 vezes mais rápido que o *Petrify*. Em domínios isolados, o *Petrigraph* superou o SatPlan por um fator de 10 vezes.

A heurística h^1 demonstrou ser menos útil ao *Petrigraph* do que era ao *Petrify*, por depender fortemente da estrutura completa da rede. É possível que existam heurísticas mais apropriadas, tais como h^1_+, que também é sugerida por Hickmott *et al* como eficiente apesar de não ser monotônica, e h^2 e afins. Uma heurística capaz de equilibrar múltiplos objetivos seria ideal. Opcionalmente, todo o algoritmo de desdobramento pode ser substituído por outros algoritmos de resolução de alcançabilidade em redes de Petri que venham a se mostrar eficientes.

O *Petrigraph* pode ser estendido com suporte a estruturas do PDDL correntemente não suportadas, tais como suporte a igualidade e axiomas, bem como extensões ao PDDL que ampliam o campo de problemas tratáveis de forma sucinta. A extensão PDDL 2.1 [34] implementa duas propriedades nessa vertente: ações durativas e métricas de plano.

A implementação de ações durativas discretas e contínuas pode ser feita adicionando à rede gerada as propriedades de rede de Petri temporal [35]. Informações temporais não alteram o fato de uma rede ser segura ou não, e assim a rede de Petri temporal pode ser desdobrada da mesma forma que uma rede Condição/Evento normal. Ações concorrentes no plano não foram consideradas neste estudo, mas são suportadas pelo algoritmo e podem trazer vantagens na análise temporal.

Métricas de plano são variáveis de estado númericas definidas na estrutura PDDL. Elas demonstram maior dificuldade aparente em serem transformadas para um equivalente em redes de Petri, porque a faixa de valores de cada métrica não é limitada pelo domínio e não tem limite dedutível, o que impossibilita criar uma rede finita equivalente. Ainda, o suporte de métricas de plano exige operações de adição, subtração, multiplicação e divisão nestas variáveis, operações que são não-triviais em valores representados com marcações numa rede de Petri, tornando uma abordagem alternativa necessária, possivelmente derivada da implementação de ações durativas. Ações durativas contínuas estão ligadas às métricas de plano. Uma implementação destas ações dependerá da resolução das situações envolvendo as métricas.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] HICKMOTT, S. L. et al. Planning via petri net unfolding. In: VELOSO, M. M. (Ed.). *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*. New York, NY, USA: Springer-Verlag, 2007. p. 1904–1911.

[2] BYLANDER, T. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, Elsevier, v. 69, n. 1-2, p. 165–204, 1994.

[3] HOWELL, R. R. The complexity of problems involving structurally bounded and conservative Petri nets. *Information Processing Letters*, Elsevier, Amsterdam, The Netherlands, The Netherlands, v. 39, n. 6, p. 309–315, 1991. ISSN 0020-0190.

[4] ESPARZA, J.; RÖMER, S.; VOGLER, W. An improvement of McMillan's unfolding algorithm. In: *TACAS*. New York, NY, USA: Springer-Verlag, 1996. (Lecture Notes in Computer Science, v. 1055), p. 87–106.

[5] EDELKAMP, S.; HELMERT, M. Exhibiting knowledge in planning problems to minimize state encoding length. In: *Recent Advances in AI Planning. 5th European Conference on Planning (ECP 99)*. New York, NY, USA: Springer-Verlag, 1999. (Lecture Notes in Artificial Intelligence, v. 1809), p. 135–147.

[6] HELMERT, M. The fast downward planning system. *Journal of Artificial Intelligence Research (JAIR)*, AAAI, v. 26, p. 191–246, 2006.

[7] MCDERMOTT, D. et al. *PDDL - The Planning Domain Definition Language*. New Haven, CN, USA, 1998.

[8] PETRI, C. A. Kommunikation mit automaten. *New York: Griffiss Air Force Base, Technical Report RADC-TR-65-377*, v. 1, p. 1–Suppl. 1, 1966. English translation.

[9] MURATA, T. Petri nets: Properties, analysis and applications. In: *Proceedings of the IEEE*. Los Alamitos, CA, USA: IEEE, 1989. v. 7, p. 541–580.

[10] CARDOSO, J.; VALETTE, R. *Redes de Petri.* Florianópolis, SC, Brasil: Editora da UFSC, 1997.

[11] LIPTON, R. J. *The Reachability Problem Requires Exponential Space.* New Haven, CN, USA, jan. 1976.

[12] ESPARZA, J. Decidability and complexity of Petri net problems - an introduction. In: REISIG, W.; ROZENBERG, G. (Ed.). *Petri Nets.* New York, NY, USA: Springer, 1996. (Lecture Notes in Computer Science, v. 1491), p. 374–428.

[13] STEWART, I. *On the Reachability Problem for Some Classes of Petri Nets.* School of Computing Science, 1991.

[14] SILVA, F. *Algoritmos para Planificação baseados em Strips.* Dissertação (Mestrado) — Universidade Federal do Paraná, Curitiba, PR, Brasil, outubro 2000.

[15] SILVA, F. *Rede de Planos: Uma proposta para a Solução de Problemas de Planejamento em Inteligência Artificial usando Redes de Petri.* Tese (Doutorado) — Centro Federal de Educação Tecnológica do Paraná - CEFET-PR, 2005.

[16] MCMILLAN, K. L. A technique of state space search based on unfolding. *Formal Methods in System Design: An International Journal*, Kluwer Academic Publishers, v. 6, n. 1, p. 45–65, January 1995.

[17] SILVA, F.; CASTILHO, M. A.; KÜNZLE, L. A. Petriplan: A new algorithm for plan generation. In: *International Joint Conference, 7th Ibero-American Conference on AI, 15th Brazilian Symposium on AI (IBERAMIA-SBIA 2000).* New York, NY, USA: Springer-Verlag, 2000. (Lecture Notes in Artificial Intelligence, 1952), p. 86–95.

[18] MONTAÑO, R. A. N. R. *Alocação de fórmulas não-clausais em planejamento com redes de Petri.* Dissertação (Programa de Pós-Graduação em Informática) — Universidade Federal do Paraná (UFPR), 2006.

[19] CARVALHO, C. S. *Algoritmos genéticos para solução de problemas de alcançabilidade em uma determinada classe de redes de petri acíclicas.* Dissertação (Mestrado) — Universidade Federal do Paraná - UFPR, 2007.

[20] BLUM, A.; FURST, M. Fast planning through planning graph analysis. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95).* New York, NY, USA: Springer-Verlag, 1995. p. 1636–1642.

[21] KAUTZ, H.; SELMAN, B. Planning as satisfiability. In: *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 92).* New York, NY, USA: John Wiley & Sons, Inc., 1992. p. 359–363.

[22] FIKES, R.; NILSSON, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. In: *Proceedings of the 2nd International Joint Conference on Artificial Intelligence (IJCAI 71).* New York, NY, USA: Springer-Verlag, 1971. p. 608–620.

[23] PAYNTER, S.; ARMSTRONG, J.; HAVEMAN, J. ADL: An activity description language for real-time networks. *Formal Aspects of Computing*, Springer London, v. 12, n. 2, p. 120–144, 2000.

[24] BONET, B.; GEFFNER, H. Planning as heuristic search. *Artificial Intelligence*, Elsevier, v. 129, n. 1-2, p. 5–33, 2001.

[25] HASLUM, P.; GEFFNER, H. Admissible heuristics for optimal planning. In: *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS 2000).* Menlo Park, CA, USA: AAAI, 2000. p. 140–149.

[26] MARYNOWSKI, J. E. *Ambiente de planejamento Ipê.* Dissertação (Mestrado) — Universidade Federal do Paraná, Curitiba, PR, Brasil, 2004.

[27] HOFFMANN, J.; NEBEL, B. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, AAAI, v. 14, p. 253–302, 2001.

[28] BACCHUS, F. The AIPS '00 planning competition. *AI Magazine*, AAAI, v. 22, n. 3, p. 47–56, 2001.

[29] LONG, D.; FOX, M. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research (JAIR)*, AAAI, v. 20, p. 1–59, 2003.

[30] GUPTA, N.; NAU, D. S. On the complexity of blocks-world planning. *Artificial Intelligence*, Elsevier, v. 56, n. 2-3, p. 223–254, 1992.

[31] KOEHLER, J.; SCHUSTER, K. Elevator control as a planning problem. In: *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS 2000)*. [S.l.]: AAAI, 2000. p. 331–338.

[32] MCDERMOTT, D. The 1998 AI planning systems competition. *AI Magazine*, AAAI, v. 21, n. 2, p. 35–55, 2000.

[33] SMITH, D.; FRANK, J.; JÓNSSON, A. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, Cambridge, v. 15, n. 1, p. 61–94, 2000.

[34] FOX, M.; LONG, D. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)*, AAAI, v. 20, p. 61–124, 2003.

[35] SUZUKI, I.; LU, H. Temporal Petri nets and their application to modeling and analysis of a handshake daisy chain arbiter. *IEEE Transactions on Computers*, IEEE, Los Alamitos, CA, USA, v. 38, n. 5, p. 696–704, 1989. ISSN 0018-9340.